

Git Community Book

The open Git resource pulled together by the whole community

AUTHORS

Thank these guys:

Alecs King (alecsk@gmail.com), Amos Waterland (apw@rossby.metr.ou.edu), Andrew Ruder (andy@aeruder.net), Andy Parkins (andyparkins@gmail.com), Arjen Laarhoven (arjen@yaph.org), Brian Hetro (whee@smaertness.net), Carl Worth (cworth@cworth.org), Christian Meder (chris@absolutegiganten.org), Dan McGee (dpmcgee@gmail.com), David Kastrup (dak@gnu.org), Dmitry V. Levin (ldv@altlinux.org), Francis Daly (francis@daoine.org), Gerrit Pape (pape@smarden.org), Greg Louis (glouis@dynamicro.ca), Gustaf Hendeby (hendeby@isy.liu.se), Horst H. von Brand (vonbrand@inf.utfsm.cl), J. Bruce Fields (bfields@fieldses.org), Jakub Narebski (jnareb@gmail.com), Jim Meyering (jim@meyering.net), Johan Herland (johan@herland.net), Johannes Schindelin (Johannes.Schindelin@gmx.de), Jon Loeliger (jdl@freescale.org), Josh Triplett (josh@freedesktop.org), Junio C Hamano (gitster@pobox.com), Linus Torvalds (torvalds@osdl.org), Lukas Sandström (lukass@etek.chalmers.se), Marcus Fritzsch (m@fritschy.de), Michael Coleman (tutufan@gmail.com), Michael Smith (msmith@cbnco.com), Mike Coleman (tutufan@gmail.com), Miklos Vajna (vmiklos@frugalware.org), Nicolas Pitre (nico@cam.org), Oliver Steele (steele@osteele.com), Paolo Ciarrocchi (paolo.ciarrocchi@gmail.com), Pavel Roskin (proski@gnu.org), Ralf Wildenhues (Ralf.Wildenhues@gmx.de), Robin Rosenberg (robin.rosenberg.lists@dewire.com), Santi Béjar (sbejar@gmail.com), Scott Chacon (schacon@gmail.com), Sergei Organov (osv@javad.com), Shawn Bohrer (shawn.bohrer@gmail.com), Shawn O. Pearce (spearce@spearce.org), Steffen Prohaska (prohaska@zib.de), Tom Prince (tom.prince@ualberta.net), William Pursell (bill.pursell@gmail.com), Yasushi SHOJI (yashi@atmark-techno.com)

MAINTAINER / EDITOR

Bug this guy:

Scott Chacon (schacon@gmail.com)

Chapter 1

Introduction

BIENVENUE SUR GIT

Bienvenue sur Git - le contrôle de version rapide et distribué.

Ce livre sert de point de départ pour gens qui veulent se familiariser à Git, et l'apprendre aussi vite et rapidement que possible.

Ce livre commencera par vous expliquer comment Git stocke les données, afin de vous montrer pourquoi c'est si différent des autres outils de contrôle de version (VCS). Cela devrait prendre environ 20 minutes.

Après nous verrons l'**Utilisation Basique de Git** - les commandes que vous utiliserez 90% du temps. Cela devrait vous donner une bonne base pour utiliser Git confortablement pour a plupart des choses que vous ferez avec. Cette section devrait vous occuper pendant environ 30 minutes de lecture.

Ensuite, nous passerons l'**Utilisation Intermédiaire de Git** - des choses un peu plus compliquées, mais qui peuvent remplacer les commandes basiques que vous aurez vu dans la première partie. Cela sera surtout des astuces et des commandes qui paraîtront plus naturelles une fois que vous connaîtrez les commandes de base.

Une fois que vous aurez maîtrisé tout ça, nous nous lancerons dans le **Git avancé** - les commandes que la plupart des gens n'utilisent pas très souvent, mais qui peuvent être très utiles dans certaines situations. Apprendre ces commandes devrait arrondir votre apprentissage de Git au jour-le-jour; vous deviendrez un maître Git!

Maintenant que vous connaissez Git, nous verrons comment **Travailler avec Git**. Ici nous parcourrons les manière d'utiliser Git dans des scripts, avec des outils de déploiement, des éditeurs et d'autres choses. Cette section est faite pour vous aider à intégrer Git dans votre environnement.

Enfin, nous auront une série d'articles sur la **documentation bas-niveau** qui peut aider les hackeurs Git qui veulent apprendre le déroulement des fonctions internes et les protocoles de Git.

Feedback et Contribution

À n'importe quel moment, si vous trouvez une erreur ou si vous voulez contribuer à ce livre, vous pouvez m'envoyer un mail à alx.girard@gmail.com, ou vous pouvez cloner le code source de ce livre à http://github.com/schacon/gitbook et m'envoyer un patch ou un pull-request.

Références

Ce livre est en grande partie tiré de différentes sources mixées et remises ensemble. Si vous voulez découvrir quelques uns des articles originaux, vous pouvez aller les voir et remercier leurs auteurs:

- Git User Manual

- The Git Tutorial
- The Git Tutorial pt 2
- "My Git Workflow" blog post

LE MODÈLE OBJET GIT

SHA

Toutes les informations nécessaires pour décrire l'historique d'un projet sont stockées dans des fichiers référencés par un "nom d'objet" de 40 caractères qui ressemble à quelque chose comme ça:

```
6ff87c4664981e4397625791c8ea3bbb5f2279a3
```

Partout dans Git, vous trouverez ces chaînes de 40 caractères. Dans chaque situation, le nom est calculé en prenant le hash SHA1 représentant le contenu de l'objet. Le hash SAH1 est une fonction de hash cryptographique. Ce que cela signifie pour nous, c'est qu'il est virtuellement impossible de trouver 2 objets différents avec le même nom. Cela à de nombreux avantages, parmi lesquels:

- Git peut rapidement savoir si 2 objets sont les mêmes ou non, juste en comparant les noms.
- Puisque le nom des objets sont calculés de la même façon dans chaque dépôt, le même contenu stocké dans des dépôts différent sera toujours stocké avec le même nom.
- Git peut détecter les erreurs quand il lit un objet, en vérifiant que le nom de l'objet est toujours le hash SHA1 de son contenu

Les Objets

Chaque objet se compose de 3 choses - un **type**, une **taille** et le **contenu**. La *taille* est simplement la taille du contenu, le contenu dépend du type de l'objet, et il y a 4 types d'objets différents: "blob", "tree", "commit", et "tag".

- Un **"blog"** est utilisé pour stocker les données d'un fichier - il s'agit en général d'un fichier.
- Un **"tree"** est comme un répertoire - il référence une liste d'autres "tree" et/ou d'autres "blobs" (i.e. fichiers et sous-répertoires).
- Un **"commit"** pointe vers un unique "tree", et le marquant afin de représenter le projet à un certain point dans le temps. Il contient des méta-informations à propos de ce point dans le temps, comme le timestamp, l'auteur du contenu depuis le dernier commit, un pointeur vers le (ou les) dernier(s) commit(s), etc.
- Un **"tag"** est une manière de représenter un commit spécifique un peu spécial. Il est normalement utilisé pour tagger certains commits en tant que version spécifique ou quelque chose comme ça.

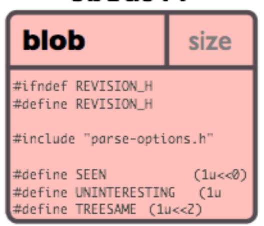
La quasi-totalité de Git est construit autour de la manipulation de cette simple structure de 4 types d'objets différents. C'est comme un mini-système de fichier qui se situe au-dessus du système de fichier de votre ordinateur.

Différences avec SVN

Il est important de noter que ce système est très différent des autres outils de contrôle de version (SCM) dont vous êtes familier. Subversion, CVS, Perforce, Mercurial et les autres utilisent tous un système de *stockage de Delta* - ils stockent les différences entre un commit et le suivant. Git ne fait pas ça - il stocke une vue instantanée de la représentation de tous les fichiers de votre projet dans une structure hiérarchisée chaque fois que vous faites un commit. C'est un concept très important pour comprendre comment utiliser Git.

L'Objet Blob

Un "blob" stocke généralement le contenu d'un fichier.



Vous pouvez utiliser git show pour examiner le contenu de n'importe quel blob. Si nous avons le SHA1 d'un blob, nous pouvons l'examiner comme ceci:

```
$ git show 6ff87c4664

 Note that the only valid version of the GPL as far as this project
 is concerned is _this_ particular version of the license (ie v2, not
 v2.2 or v3.x or whatever), unless explicitly otherwise stated.
...
```

Un "blog" n'est rien de plus qu'un morceau de données binaires. Il ne fait référence à rien, et n'a aucun attribut, même pas un nom de fichier.

Puisque le blob est entièrement défini par ses données, si 2 fichiers dans un répertoire (ou dans différentes versions du dépôt) ont le même contenu, ils partageront alors le même objet blob. Cet objet est totalement indépendant de l'endroit où il se trouve dans la hiérarchie des dossiers, et renommer un fichier ne change pas l'objet auquel ce fichier est associé.

L'Objet Tree

Un "tree" est un simple objet qui contient une liste de pointeurs vers des "blobs" et d'autres "trees" - il représente généralement le contenu d'un répertoire ou sous-répertoire.

c36d4..

tree		size
blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff

La commande polyvalente git show peut aussi Être utilisée pour examiner un objet "tree", mais git ls-tree vous donnera plus de détails. Si nous avons le SHA1 d'un tree, nous pouvons le détailler comme ceci:

```
$ git ls-tree fb3a8bdd0ce
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c    .gitignore
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d    .mailmap
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745    Documentation
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200    GIT-VERSION-GEN
100644 blob 289b046a443c0647624607d471289b2c7dcd470b    INSTALL
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1    Makefile
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52    README
...
```

Comme vous pouvez le voir, un objet tree contient une liste d'entrées, chacune avec un mode, un type d'objet, un nom SHA1, un nom, le tout trié avec le nom. L'objet tree représente le contenu d'un unique dossier.

Un objet référencé par un tree peut être un blog, représentant le contenu d'un fichier, ou un autre tree, représentant le contenu d'un sous-répertoire. Puisque les trees et les blobs, comme les autres objets, sont nommés par le hash SHA1 de leur contenu, 2 trees ont le même nom SHA1 si, et seulement si, leur contenu (en incluant, récursivement, le contenu de tous les sous-répertoires) est identique. Cela permet à git de déterminer rapidement les différences entre 2 objets trees associés, puisqu'il peut ignorer les entrées avec le même nom d'objet.

(Note: en présence de sous-modules, les trees peuvent aussi contenir des commits comme entrées. Voir la section **Sous-Modules**.)

Notez que tous les fichiers ont le mode 644 ou 755: git ne tient compte que du bit exécutable.

L'Objet Commit

L'objet "commit" lie l'état physique d'un "tree" avec une description de la manière et de la raison de l'arrivée à cet état.



Vous pouvez utiliser git show ou git log avec l'option --pretty=raw pour examiner vos commits favoris:

```
$ git show -s --pretty=raw 2be7fcb476
commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
parent 257a84d9d02e90447b149af58b271c19405edb6a
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700

    Fix misspelling of 'suppress' in docs

    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

Comme vous pouvez le voir, un commit est défini par:

- Un **tree**: Le nom SHA1 de l'objet "tree" (comme défini précédemment), représentant le contenu d'un répertoire à un certain moment.
- **parent(s)**: Le nom SHA1 de (ou des) numéro de commits qui représente l'étape antérieure dans l'historique du projet. L'exemple ci-dessus a un parent; les commits mergés peuvent en avoir plus d'un. Un commit sans parent est nommé le commit "racine", et représente la révision initiale d'un projet. Chaque projet doit contenir au moins une racine. Un projet peut avoir plusieurs racine, bien que ça ne soit pas très commun (ou que ça ne soit pas une bonne idée).
- Un **author**: Le nom de la personne responsable de ce changement, avec sa date
- Un **committer**: Le nom de la personne qui a créé le commit, avec la date de création. Cet attribut peut être différent de l'auteur; par exemple, si l'auteur écrit un patch est l'envoi à une autre personne par mail, cet personne peut utilisé le patch pour créer le commit.
- Un **comment** qui décrit ce commit.

Notez qu'un commit ne contient pas d'information à propos de ce qui a été modifié; tous les changements sont calculés en comparant les contenus du "tree" référencé dans ce commit avec le "tree" associé au(x) parent(s) du commit. En particulier, git n'essaye pas d'enregistrer le rennomage de fichier explicitement, bien qu'il puisse identifier des cas où la

persistance des données d'un fichier avec un chemin modifié suggère un rennomage. (Voir, par exemple, la commande git diff avec l'option -M).

Un commit est normalement créé avec la commande git commit, qui crée un commit dont le parent est est le HEAD courant, et avec le "tree" pris depuis le contenu actuellement stocké dans l'index.

Le Modèle Objet

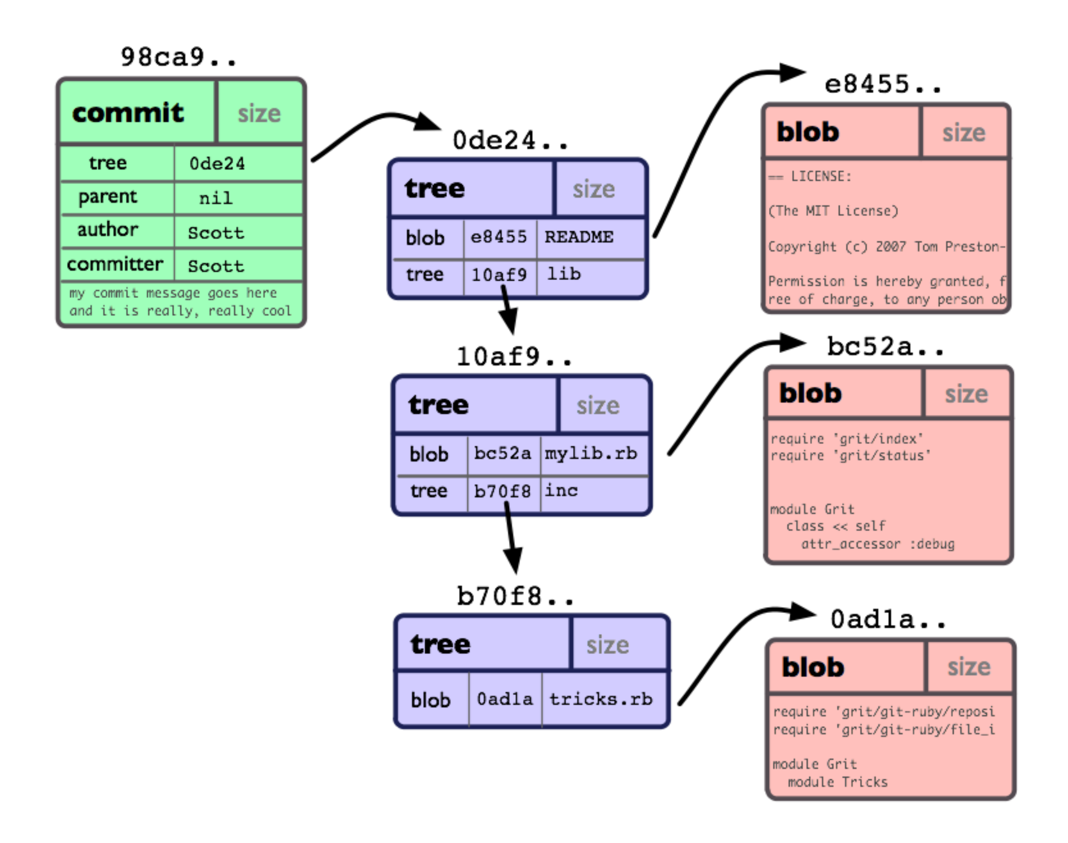
Donc, maintenant que nous avons vu les 3 types d'objets principaux (blob, tree et commit), regardons rapidement comment ils travaillent ensemble.

Si nous avons un simple projet avec la structure de dossiers suivante:

```
$>tree
.
|-- README
`-- lib
    |-- inc
    |   `-- tricks.rb
    `-- mylib.rb

2 directories, 3 files
```

Et si nous committons ce projet sur un dépôt Git, il sera représenté comme ça:



Vous pouvez voir que nous avons créé un objet **tree** pour chaque répertoire (pour la racine aussi) et un objet **blob** pour chaque fichier. Ensuite nous avons un objet **commit** qui pointe vers la racine, afin que nous puissions récupérer l'apparence du projet quand il a été committé.

L'Objet Tag

49e11..

tag | size

object ae668

type commit

tagger Scott

my tag message that explains this tag

Un objet "tag" contient un nom d'objet (simplement nommé 'object'), un type d'objet, un nom de tag, le nom de la personne ("taggeur") qui a créé le tag, et un message, qui peut contenir une signature, comme on peut le voir en utilisant git cat-file:

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----
```

Voyez la commande git tag pour apprendre comment créer et vérifier les objets "tags". (Notez que git tag peut aussi être utilisé pour créer des "tags légers", qui ne sont pas du tout des objets "tags", mais juste des simples références dont le nom commence par "refs/tags/").

LE RÉPERTOIRE GIT ET LE RÉPERTOIRE DE TRAVAIL

Le Répertoire Git

Le 'répertoire git' est un répertoire qui contient tout l'historique de Git et les méta-informations du projet - avec tous les objets (commits, tree, blobs, tags), tous les pointeurs vers les différentes branches et plus encore.

Il y a un seul répertoire Git par projet (au contraire de système comme SVN ou CVS qui contient ce répertoire dans chaque sous-répertoire du projet), ce dossier se nomme '.git' (par défaut, il peut être nommé différemment), et ce dossier se situe à la racine de votre projet. Si vous regardez le contenu de ce répertoire, vous pouvez y trouver tous vos fichiers important:

```
$>tree -L 1
.
|-- HEAD         # pointeur vers votre branche courante
|-- config       # configuration de vos préférences
|-- description  # description de votre projet
|-- hooks/       # pre/post action hooks
|-- index        # fichier d'index (voir prochaine section)
|-- logs/        # un historique de votre branche
|-- objects/     # vos objets (commits, trees, blobs, tags)
`-- refs/        # pointeurs vers vos branches
```

(Vous pourrez trouver d'autres fichiers/dossiers ici aussi, mais ils ne sont pas important pour l'instant)

Le Répertoire de Travail

Le 'répertoire de travail' de Git est le répertoire qui contient la version courante des fichiers sur lesquels vous travaillez. Les fichiers de ce répertoire sont souvent effacés ou remplacés par Git quand vous changez de branche - c'est tout à fait normal. Tout votre historique est stocké dans votre répertoire Git; le répertoire de travail est simplement une version temporaire de votre projet où vous modifier les fichiers jusqu'à votre prochain commit.

L'INDEX GIT

L'index Git est une zone d'assemblage entre votre répertoire de travail et votre dépôt. Vous pouvez utiliser l'index pour construire un groupe de changements qui seront committé ensemble. Quand vous créez un commit, ce qui se trouve dans l'index est ce qui est committé, et pas ce qui se trouve dans le répertoire de travail.

A l'Intérieur de l'Index

La façon la plus simple de voir ce qu'est l'index est d'utiliser la commande git status. Quand vous lancez git status, vous pouvez voir quels fichiers sont assemblés (actuellement dans l'index), quels sont ceux modifiés mais pas assemblés, et ceux qui ne sont pas suivis.

```
$>git status
# On branch master
# Your branch is behind 'origin/master' by 11 commits, and can be fast-forwarded.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   daemon.c
#
```

```
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   grep.c
#   modified:   grep.h
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   blametree
#   blametree-init
#   git-gui/git-citool
```

Si vous effacé complètement l'index, vous ne perdrez généralement aucune information tant que vous avez le nom du tree qui est décrit.

Et avec ça, vous devriez avoir un bonne compréhension des bases de ce que Git fait en arrière plan, et pourquoi il est différent de la plupart des autres systèmes de contrôle de version. Ne vous inquiétez pas si vous n'avez pas encore tout compris; nous reviendrons sur tous ces point dans les prochaines parties du livre: Maintenant, nous sommes prêts à installer, configurer et utiliser Git.

Chapter 2

Le Commencement

INSTALLER GIT

Installer depuis le Code Source

Rapidement, sur un système Unix, vous pouvez télécharger le code source de Git depuis la Page de Téléchargement de Git, puis dérouler l'installation de cette façon:

```
$ make prefix=/usr all ;# avec votre utilisateur
$ make prefix=/usr install ;# avec l'utilisateur root
```

Vous aurez besoin des librairies expat, curl, zlib, et openssl installées - normalement, à part *expat*, ces librairies devraient déjà se trouver sur votre système.

Linux

Si vous utilisez Linux, vous devez pouvoir installer Git facilement depuis le gestionnaire de paquet de votre système:

```
$ yum install git-core
```

```
$ apt-get install git-core
```

Si cela ne fonctionne pas, vous pouvez télécharger les paquets .deb ou .rpm à ces adresses:

RPM Packages

Stable Debs

Si vous préférer installer Git depuis le code source sur votre système Linux, cet article pourra vous aider:

Article: Installing Git on Ubuntu

Mac 10.4

Pour Mac 10.4 et 10.5, vous pouvez installer Git via MacPorts, s'il est déjà installé. Sinon, vous pouvez l'installer depuis ici.

Une fois que MacPorts est installé, il vous suffit de faire ceci:

```
$ sudo port install git-core
```

Si vous préférez installer Git depuis le code source, ces articles peuvent vous aider:

Article: Installing Git on Tiger

Article: Installing Git and git-svn on Tiger from source

Mac 10.5

Avec Leopard, vous pouvez installer Git via MacPorts, mais vus avez une autre option qui vous permet d'installer Git avec une interface d'installation. Vous pouvez télécharger cette interface ici: Git OSX Installer

Si vous préférez installer Git depuis le code source, ces articles peuvent vous aider:

Article: Installing Git on OSX Leopard

Article: Installing Git on OS 10.5

Windows

Sur Windows, il est assz simple d'installe Git. Téléchargez et installez simplement le paquet msysGit.

Allez au chapitre *Git sur Windows* pour regarder un screencast qui vous fera une démonstration de l'installation et de l'usage de Git sur Windows.

CONFIGURATION ET INITIALISATION

Configuration de Git

La première chose que vous voudrez et devrez faire, c'est de configurer votre nom et votre adresse mail que Git utilisera quand vous signerez vos commits.

```
$ git config --global user.name "Scott Chacon"
$ git config --global user.email "schacon@gmail.com"
```

Ces commandes vont créer un fichier dans votre répertoire utilisateur qui sera utilisé par tout vos projets. Par défaut, ce fichier est *~/.gitconfig* et il contiendra quelque chose comme ça:

```
[user]
        name = Scott Chacon
        email = schacon@gmail.com
```

Si vous voulez changez ces valeurs pour un projet spécifique (pour utiliser un mail d'entreprise par exemple), vous pouvez lancer la commande *git config*, sans l'option *--global*, quand vous êtes à l'intérieur de votre projet. Cela ajoutera une section [user], comme celle que nous venons de voir, dans le fichier *.git/config* qui se trouve à la racine de votre projet.

Chapter 3

Utilisation Basique

OBTENIR UN DÉPÔT GIT

Maintenant que nous avons tout configuré, nous avons besoin d'un dépôt Git. Nous pouvons faire ça de 2 manière - soit *clonant* un dépôt qui existe déjà, soit en *initialisant* un dépôt depuis un dossier vide ou depuis des fichiers existant qui ne sont pas encore sous contrôle de version.

Cloner un Dépôt

Afin d'obtenir une copie d'un projet, vous aurez besoin de connaître l'adresse URL Git du projet - l'endroit où se trouve le dépôt. Git peut être utilisé avec de nombreux protocoles, donc cette adresse peut commencer avec ssh://, http(s)://, git://, ou juste un nom d'utilisateur (en supposant que git passe par ssh). Par exemple, le code source de Git même peut être cloné en passant par le protocole git:

```
git clone git://git.kernel.org/pub/scm/git/git.git
```

ou par http:

```
git clone http://www.kernel.org/pub/scm/git/git.git
```

Le protocole git:// est plus rapide et plus efficace, mais il est parfois nécessaire d'utiliser http derrière un firewall d'entreprise par exemple. Dans tous les cas, vous devriez maintenant avoir un répertoire nommé 'git' qui contient tout le code source de Git et son historique - c'est simplement une copie de ce qui se trouvait sur le serveur.

Par défaut, Git nommera le nouveau répertoire, où il stockera votre code cloné, en prenant ce qui arrive juste avant le '.git' dans le chemin du projet cloné. (ie. *git clone http://git.kernel.org/linux/kernel/git/torvalds/linux-2.6.git* créera un nouveau répertoire nommé 'linux-2.6' pour y cloner le code).

Initialiser un Nouveau Dépôt

Imaginons que nous avons une archive nommée project.tar.gz avec notre travail initial. Nous pouvons le placé sous le contrôle de version de Git comme ceci:

```
$ tar xzf project.tar.gz
$ cd project
$ git init
```

Git vous répondra:

```
Initialized empty Git repository in .git/
```

Vus avez maintenant initialisé le répertoire de travail--vous pourrez y trouver un nouveau répertoire à l'intérieur, nommé ".git".

WORKFLOW NORMAL

Modifiez quelques fichiers, puis ajoutez leur contenu mis à jour dans l'index:

```
$ git add file1 file2 file3
```

Vous être maintenant prêts pour le commit. Vous pouvez vérifier ce qui va être committé en utilisant la commande git diff avec l'option --cached:

```
$ git diff --cached
```

(Sans l'option --cached, git diff vous montrera les changements qui vous avez fait mais que vous n'avait pas encore ajouté à l'index.) Vous pouvez aussi trouver un résumé rapide de la situation en utilisant la commande git status:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   file1
#   modified:   file2
#   modified:   file3
#
```

Si vous devez faire d'autres changements, faites le maintenant, puis ajoutez le contenu modifié à l'index. Enfin, committez vos changements comme ceci:

```
$ git commit
```

Il vous demandera de laisser un message pour décrire les changements qui ont eu lieu, puis enregistrera une nouvelle version du projet.

Au lieu de lancer `git add` avant vos commit, vous pouvez aussi utiliser directement cette commande:

```
$ git commit -a
```

Le commit trouvera automatiquement les fichiers modifiés (mais pas les nouveaux), les ajoutera à l'index et les committera, tout ça dans foulée.

Une note sur le message du commit: Bien que ça ne soit pas obligatoire, c'est une bonne idée de commencer le message du commit avec une courte ligne (moins de 50 caractères) qui résume le changement, suivi d'une ligne blanche puis d'une description plus complète. Les outils qui transforment les commits en mail, par exemple, utilisent la première ligne du commit pour le Sujet du mail, et le reste pour le contenu.

Git surveille le contenu et pas les fichiers

Beaucoup de système de contrôle de version fournisse une commande "add" qui demande au système de surveiller les changements dans un nouveau fichier. La commande "add" de Git fait quelque chose de plus simple et de plus puissant: `git add` est utilisé à la fois pour les nouveaux fichiers et les fichiers nouvellement modifiés. Dans les 2 cas, il prend une capture des fichiers fournis et assemble leur contenu dans l'index, prêt à être inclus dans le prochain commit.

USAGE BASIQUE DES BRANCHES ET DES MERGES

Un seul dépôt git peut maintenir de nombreuses branches de développement. Pour créer une nouvelle branche nommée "experimental", utilisez:

```
$ git branch experimental
```

Si vous lancez maintenant

```
$ git branch
```

Vous obtiendrez la liste de toutes les branches existantes:

```
  experimental
* master
```

La branche "experimental" est celle que vous venez de créer, et la branche "master" est la branche par défaut qui a été créée automatiquement pour vous. L'astérisque signale la branche su laquelle vous vous trouvez. Tapez:

```
$ git checkout experimental
```

pour passer sur la branche "experimental". Éditez maintenant un fichier, committer le changement, et revenez sur la branche "master":

```
(éditer un fichier)
$ git commit -a
$ git checkout master
```

Vous pouvez vérifier que le changement que vous venez de faire n'est plus visible, puisqu'il a été fait sur la branche "experimental" et que nous sommes revenus sur la branche "master".

Vous pouvez faire un changement différent sur la branche "master":

```
(éditer un fichier)
$ git commit -a
```

à cet instant, les 2 branches ont divergé, avec des changements différents faits dans chacune d'elles. Pour fusionner (merger) les changements effectués dans la branche "experimental" sur la branche "master", lancez:

```
$ git merge experimental
```

Si les changements ne créent pas de conflit, vous avez fini. S'il y a des conflits, un marquage sera laissé dans les fichiers problématiques pour vous montrer le conflit:

```
$ git diff
```

vous montrera ces marquages. Une fois que vous aurez édité les fichiers pour résoudre les conflits,

```
$ git commit -a
```

committera le résultat du merge. Enfin,

```
$ gitk
```

vous montrera une jolie représentation graphique de l'historique obtenu.

Maintenant, vous pouvez effacer la branche "experimental" avec

```
$ git branch -d experimental
```

Cette commande s'assure que les changement de la branche "experimental" se trouve dans la branche courante.

Si vous développez une idée folle, puis le regrettez, vous pouvez toujours effacer cette branche avec:

```
$ git branch -D crazy-idea
```

Les branches sont faciles et peu coûteuse, donc c'est un bon moyen pour tester des choses nouvelles.

Comment merger

Vous pouvez joindre 2 branches de développement divergentes en utilisant git merge:

```
$ git merge titrebranche
```

merge les changements fait dans la branche "titrebranche" dans la branche courante. Si il y a des conflits--par exemple, si le même fichier est modifié au même endroit de 2 façons différente dans la branche distante et la branche locale--vous serez avertis; l'avertissement peut ressembler à quelque chose comme ça:

```
$ git merge next
 100% (4/4) done
Auto-merged file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

des marqueurs de conflit sont ajoutés aux fichiers problématiques, et après avoir résolu les conflits manuellement, vous pouvez mettre à jour l'index avec le nouveau contenu et lancer `git commit`, comme vous le feriez quand vous modifiez un fichier.

Si vous analysez le résultat de ce commit avec gitk, vous verrez qu'il à 2 parents: l'un pointant vers le sommet de la branche courante, et l'autre pointant vers le sommet de l'autre branche.

Résoudre un merge

Quand un merge n'est pas résolu automatiquement, git laisse l'index et le "tree" de travail dans un état spécial vous donnant toutes les informations dont vous aurez besoin pour vous aider à résoudre le merge.

Les fichiers en conflits sont marqués spécialement dans l'index, donc jusqu'à que vous ayez résolu le problème et mis à jour l'index, git commit ne fonctionnera pas:

```
$ git commit
file.txt: needs merge
```

De plus, git status vous donnera la liste de ces fichiers "non-mergés", et les fichiers contenant des conflits auront ces conflits marqués comme ceci:

```
<<<<<<< HEAD:file.txt
Hello world
=======
Goodbye
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

Vous n'avez plus qu'à éditer ces fichier pour résoudre les conflits, puis:

```
$ git add file.txt
$ git commit
```

Notez que le message du commit sera déjà rempli pour vous avec quelques informations à propos du merge. Normalement vous pouvez juste laisser ce message inchangé, mais vous voudrez peut être ajouter un commentaire additionnel si vous les désirez.

Cette partie contient tout ce que vous avec besoin de savoir pour résoudre un merge simple. Mais git vous fournira aussi plus d'information pour vous aider à résoudre les conflits:

Annuler un merge

Si vous êtes bloqués et décidés de laisser tomber en jetant tout vos brouillons par la fenêtre, vous pouvez toujours revenir à l'état où vous vous trouviez avant le merge avec la commande:

```
$ git reset --hard HEAD
```

Ou, si vous avez déjà committé le merge que vous voulez balancer,

```
$ git reset --hard ORIG_HEAD
```

Cependant, cette dernière commande peut être dangereuse dans certains cas--ne balancez jamais un commit si ce commit était lui même le merge d'une autre branche, en faisant ça vous risqueriez de rendre confus les prochains merges.

Avance rapide des merges

Il y a un cas spécial non-mentionné plus tôt, qui est traité différemment. Normalement, un merge est un commit avec 2 parents, un pour chacune des 2 lignes de développement qui seront mergées.

Cependant, si la branche courante n'a pas divergé de l'autres--donc tous les commit présent dans la branche courant sont déjà contenus dans l'autre branche--alors git ne fait qu'une "avance rapide"; le sommet (head) de la branche courante est avancé jusqu'au point du sommet de la branche à merger, sans qu'aucun commit ne soit créé.

ANALYSER L'HISTORIQUE - GIT LOG

La commande git log vous montrera des listes de commits. Tout seul, elle vous montrera les commits visible depuis le commit parent de votre version courante; mais vous pouvez faire des requêtes plus spécifiques:

```
$ git log v2.5..        # commits depuis (non-visible depuis) v2.5
$ git log test..master  # commits visibles depuis master mais pas test
$ git log master..test  # commits visibles depuis test mais pas master
$ git log master...test # commits visibles pour test ou
                        #    master, mais pas les 2
$ git log --since="2 weeks ago" # commits des 2 dernières semaines
$ git log Makefile      # commits qui modifient le Makefile
$ git log fs/           # commits qui modifient les fichiers sous fs/
$ git log -S'foo()'     # commits qui ajoutent ou effacent des données
                        #    contenant la chaîne 'foo()'
$ git log --no-merges   # ne pas montrer les commits de merge
```

Et vous pouvez bien sûr combiner toutes ces options; la requête suivante trouve tous les commits depuis v2.5 qui touchent le Makefile et tous les fichier sous fs/:

```
$ git log v2.5.. Makefile fs/
```

Git log vous montrera la liste de chaque commit, le plus récent d'abord, qui correspond aux arguments donnés à la commande de log.

```
commit f491239170cb1463c7c3cd970862d6de636ba787
Author: Matt McCutchen <matt@mattmccutchen.net>
Date:   Thu Aug 14 13:37:41 2008 -0400

    git format-patch documentation: clarify what --cover-letter does

commit 7950659dc9ef7f2b50b18010622299c508bfdfc3
```

```
Author: Eric Raible <raible@gmail.com>
Date:   Thu Aug 14 10:12:54 2008 -0700

    bash completion: 'git apply' should use 'fix' not 'strip'
    Bring completion up to date with the man page.
```

Vous pouvez aussi demander à git de vous montrer les patches:

```
$ git log -p

commit da9973c6f9600d90e64aac647f3ed22dfd692f70
Author: Robert Schiele <rschiele@gmail.com>
Date:   Mon Aug 18 16:17:04 2008 +0200

    adapt git-cvsserver manpage to dash-free syntax

diff --git a/Documentation/git-cvsserver.txt b/Documentation/git-cvsserver.txt
index c2d3c90..785779e 100644
--- a/Documentation/git-cvsserver.txt
+++ b/Documentation/git-cvsserver.txt
@@ -11,7 +11,7 @@ SYNOPSIS
 SSH:

 [verse]
-export CVS_SERVER=git-cvsserver
+export CVS_SERVER="git cvsserver"
 'cvs' -d :ext:user@server/path/repo.git co <HEAD_name>

 pserver (/etc/inetd.conf):
```

Statistique sur les Log

Vous pouvez rajouter l'options `--stat` à 'git log' pour obtenir la liste des fichiers qui ont été changés dans un commit et afficher combien de lignes ont été rajouté ou retiré de chaque fichier.

```
$ git log --stat

commit dba9194a49452b5f093b96872e19c91b50e526aa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Sun Aug 17 15:44:11 2008 -0700

    Start 1.6.0.X maintenance series

 Documentation/RelNotes-1.6.0.1.txt |   15 +++++++++++++++
 RelNotes                           |    2 +-
 2 files changed, 16 insertions(+), 1 deletions(-)
```

Formater le Log

Vous pouvez formater la sortie du log pour afficher ce que vous voulez. L'option '--pretty' peut servir pour afficher les log avec un format pré-réglé, comme 'oneline':

```
$ git log --pretty=oneline
a6b444f570558a5f31ab508dc2a24dc34773825f dammit, this is the second time this has reverted
49d77f72783e4e9f12d1bbcacc45e7a15c800240 modified index to create refs/heads if it is not
9764edd90cf9a423c9698a2f1e814f16f0111238 Add diff-lcs dependency
e1ba1e3ca83d53a2f16b39c453fad33380f8d1cc Add dependency for Open4
0f87b4d9020fff756c18323106b3fd4e2f422135 merged recent changes: * accepts relative alt pat
f0ce7d5979dfb0f415799d086e14a8d2f9653300 updated the Manifest file
```

Ou vous pouvez utiliser le format 'short':

```
$ git log --pretty=short
commit a6b444f570558a5f31ab508dc2a24dc34773825f
Author: Scott Chacon <schacon@gmail.com>

    dammit, this is the second time this has reverted

commit 49d77f72783e4e9f12d1bbcacc45e7a15c800240
Author: Scott Chacon <schacon@gmail.com>

    modified index to create refs/heads if it is not there

commit 9764edd90cf9a423c9698a2f1e814f16f0111238
Author: Hans Engel <engel@engel.uk.to>

    Add diff-lcs dependency
```

Vous pouvez aussi utiliser 'medium', 'full', 'fuller', 'email' ou 'raw'. Si ces format ne sont pas exactement ce dont vous avez besoin, vous pouvez créer votre propre format avec l'option '--pretty=format' (voir la documentation de git log) pour toutes les options de formatage.

```
$ git log --pretty=format:'%h was %an, %ar, message: %s'
a6b444f was Scott Chacon, 5 days ago, message: dammit, this is the second time this has re
49d77f7 was Scott Chacon, 8 days ago, message: modified index to create refs/heads if it i
9764edd was Hans Engel, 11 days ago, message: Add diff-lcs dependency
e1ba1e3 was Hans Engel, 11 days ago, message: Add dependency for Open4
0f87b4d was Scott Chacon, 12 days ago, message: merged recent changes:
```

Autre chose intéressante, vous pouvez aussi visualiser le graphe des commits avec l'options '--graph', comme ceci:

```
$ git log --pretty=format:'%h : %s' --graph
* 2d3acf9 : ignore errors from SIGCHLD on trap
*   5e3ee11 : Merge branch 'master' of git://github.com/dustin/grit
|\
```

```
| * 420eac9 : Added a method for getting the current branch.
* | 30e367c : timeout code and tests
* | 5a09431 : add timeout protection to grit
* | e1193f8 : support for heads with slashes in them
|/
* d6016bc : require time for xmlschema
```

Ça vous montrera un représentation plutôt réussie des lignes de l'historique des commits.

Ordonner le Log

Vous pouvez aussi voir les entrées de log dans des ordres différents. Notez que git log commence par le commit le plus récent et va à reculons vers ses parents; cependant, puisque l'historique de git peut contenir de multiples lignes indépendantes de développement, l'ordre de l'affiche des commits est plutôt arbitraire.

Si vous voulez spécifier un ordre en particulier, vous pouvez ajouter une option à la commande git log.

Pa défaut les commit sont montrés dans l'ordre inversement chronologique.

Cependant, si vous ajoutez l'option '--topo-order', les commits apparaîtrons dans l'ordre topologique (i.e. les commits descendants sont affichés avant leurs parents). Si nous regardons le git log pour le dépôt de Grit dans un ordre topologique, vous pouvez voir que toutes les lignes de développement sont regroupées ensembles.

```
$ git log --pretty=format:'%h : %s' --topo-order --graph
*   4a904d7 : Merge branch 'idx2'
|\
| *   dfeffce : merged in bryces changes and fixed some testing issues
| |\
| | * 23f4ecf : Clarify how to get a full count out of Repo#commits
| | *   9d6d250 : Appropriate time-zone test fix from halorgium
```

```
| | |\
| | | * cec36f7 : Fix the to_hash test to run in US/Pacific time
| | * | decfe7b : fixed manifest and grit.rb to make correct gemspec
| | * | cd27d57 : added lib/grit/commit_stats.rb to the big list o' files
| | * | 823a9d9 : cleared out errors by adding in Grit::Git#run method
| | * |   4eb3bf0 : resolved merge conflicts, hopefully amicably
| | |\ \
| | | * | d065e76 : empty commit to push project to runcoderun
| | | * | 3fa3284 : whitespace
| | | * | d01cffd : whitespace
| | | * | 7c74272 : oops, update version here too
| | | * | 13f8cc3 : push 0.8.3
| | | * | 06bae5a : capture stderr and log it if debug is true when running commands
| | | * | 0b5bedf : update history
| | | * | d40e1f0 : some docs
| | | * | ef8a23c : update gemspec to include the newly added files to manifest
| | | * | 15dd347 : add missing files to manifest; add grit test
| | | * | 3dabb6a : allow sending debug messages to a user defined logger if provided; tes
| | | * | eac1c37 : pull out the date in this assertion and compare as xmlschemaw, to avoi
| | | * | 0a7d387 : Removed debug print.
| | | * | 4d6b69c : Fixed to close opened file description.
```

Vous pouvez aussi utiliser '--date-order', qui ordonne les commits par date. Cette option est similaire à '--topo-order' dans le sens où les parents seront affichés après tus leurs enfants, mais autrement les commits sont toujours ordonnés suivant la date. Vous pouvez voir ici que les lignes de développement sont groupées ensembles, et qu'elle s'éloignent quand un développement parallèle à lieu:

```
$ git log --pretty=format:'%h : %s' --date-order --graph
*   4a904d7 : Merge branch 'idx2'
|\
* | 81a3e0d : updated packfile code to recognize index v2
| *   dfeffce : merged in bryces changes and fixed some testing issues
| |\
```

```
| * | c615d80 : fixed a log issue
|/ /
| * 23f4ecf : Clarify how to get a full count out of Repo#commits
| *   9d6d250 : Appropriate time-zone test fix from halorgium
| |\
| * | decfe7b : fixed manifest and grit.rb to make correct gemspec
| * | cd27d57 : added lib/grit/commit_stats.rb to the big list o' file
| * | 823a9d9 : cleared out errors by adding in Grit::Git#run method
| * |   4eb3bf0 : resolved merge conflicts, hopefully amicably
| |\ \
| * | | ba23640 : Fix CommitDb errors in test (was this the right fix?
| * | | 4d8873e : test_commit no longer fails if you're not in PDT
| * | | b3285ad : Use the appropriate method to find a first occurrenc
| * | | 44dda6c : more cleanly accept separate options for initializin
| * | | 839ba9f : needed to be able to ask Repo.new to work with a bar
| | * | d065e76 : empty commit to push project to runcoderun
* | | | 791ec6b : updated grit gemspec
* | | | 756a947 : including code from github updates
| | * | 3fa3284 : whitespace
| | * | d01cffd : whitespace
| * | | a0e4a3d : updated grit gemspec
| * | | 7569d0d : including code from github updates
```

Enfin, vous pouvez inverser l'ordre du log avec l'option '--reverse'.

COMPARER LES COMMITS - GIT DIFF

Vous pouvez générer les différences entres 2 version de votre projet en utilisant git diff:

```
$ git diff master..test
```

Cette commande produit une différence entre le sommet de 2 banches. Si vous préférez trouver la différence de leur ancêtre commun, vous pouvez utiliser 3 points au lieu de 2:

```
$ git diff master...test
```

git diff est un outils incroyablement utile pour trouver ce qui a changé entre 2 points dans l'historique de votre projet, ou pour voir quelle personne a essayé d'introduire une nouvelle branche, etc.

Ce que vous committerez

Vous utiliserez couramment git diff pour trouver les différences entre votre dernier commit, votre index, et votre répertoire de travail courant. Un usage courant est de lancer:

```
$ git diff
```

ce qui vous montrera les changements dans le répertoire de travail qui ne sont pas encore assemblés pour le prochain commit. Si vous voulez voir ce qui *est* assemblé pour le prochain commit, vous pouvez lancer:

```
$ git diff --cached
```

ce qui vous montrera la différence entre l'index et votre dernier commit; ce que vous committerez si vous lancez "git commit" sans l'option "-a". Enfin, vous pouvez lancer

```
$ git diff HEAD
```

pour afficher les changements de votre le répertoire de travail depuis votre dernier commit; ces changements seront committés si vous lancez "git commit -a".

Plus d'Options de Diff

Si vous voulez voir comment votre répertoire de travail actuel diffère de l'état du projet dans une autre branche, vous pouvez lancer quelque chose comme ça:

```
$ git diff test
```

Cela vous montrera la différence entre votre répertoire de travail actuel et la capture de la branche 'test'. Vous pouvez aussi limiter la différence à un fichier spécifique ou à un sous-répertoire en ajoutant un *limiteur de chemin*:

```
$ git diff HEAD -- ./lib
```

Cette commande vous montrera les différences entre votre répertoire de travail actuel et le dernier commit (ou , plus précisément, le sommet de la branche actuelle), en limitant la comparaison aux fichiers dans le répertoire 'lib'.

Si vous ne voulez pas voir le patch complet, vous pouvez ajouter l'option '--stat', qui limitera la sortie aux noms de fichier qui ont changés, accompagné d'un petit graphe décrivant le nombre de lignes différentes dans chaque fichier.

```
$>git diff --stat
 layout/book_index_template.html                    |    8 ++-
 text/05_Installing_Git/0_Source.markdown           |   14 ++++++
 text/05_Installing_Git/1_Linux.markdown            |   17 +++++++
 text/05_Installing_Git/2_Mac_104.markdown          |   11 +++++
 text/05_Installing_Git/3_Mac_105.markdown          |    8 ++++
 text/05_Installing_Git/4_Windows.markdown          |    7 +++
 .../1_Getting_a_Git_Repo.markdown                  |    7 +++-
 .../0_ Comparing_Commits_Git_Diff.markdown         |   45 +++++++++++++++++++-
 .../0_ Hosting_Git_gitweb_repoorcz_github.markdown |    4 +-
 9 files changed, 115 insertions(+), 6 deletions(-)
```

Cela permet parfois de voir plus facilement les changements effectués

WORKFLOWS DISTRIBUÉS

Supposons qu'Alice a démarré un nouveau projet dans son dépôt git situé dans /home/alice/project, et que Bob, qui a un répertoire utilisateur sur la même machine (/home/bob/), veuille y contribuer.

Bob commence par:

```
$ git clone /home/alice/project mondepot
```

Cela crée un répertoire "mondepot" qui contient un clone du dépôt d'Alice. Le clone est une copie parfaite du projet original, possédant aussi sa propre copie de l'historique du projet original.

Bob fait quelques changements et les commit:

```
(editer des fichiers)
$ git commit -a
(répéter autant que nécessaire)
```

Quand il est prêt, il dit à Alice de récupérer (pull) ses changements depuis son dépôt situé dans /home/bob/mondepot. Alice fait alors:

```
$ cd /home/alice/project
$ git pull /home/bob/myrepo master
```

Cela merge les changement de la branche "master" de Bob dans la branche courante d'Alice. Si Alice a fait ses propres changements pendant ce temps, alors elle devra peut être réparer quelques conflit à la main. (Notes l'option "master" dans la commande ci-dessus, elle n'est pas nécessaire car c'est l'option par défaut.)

La commande "pull" travaille donc en 2 étapes: elle récupère les changements d'une branche distante, et merge ces changements dans la branche courante.

Quand vous travaillez dans une petite équipe soudée, il est courant que tous interagissent très souvent avec le même dépôt. En définissant un raccourci pour le dépôt 'distant', nous pouvons rendre ces opération plus simples:

```
$ git remote add bob /home/bob/mondepot
```

Avec ça, Alice peut effectuer la première opération seulement en utilisant la commande "git fetch", elle ne mergera pas les modifications avec sa propre branche:

```
$ git fetch bob
```

Contrairement à la version longue, quand Alice récupère (fetch) les données de Bob en utilisant un raccourci configuré avec 'git remote'. alors ce qui est récupéré est stocké dans une branche de suivi distant, dans notre cas 'bob/master'. Donc maintenant:

```
$ git log -p master..bob/master
```

montre la liste de tous les changements que Bob a fait depuis qu'il a créé une branche depuis la branche "master" d'Alice.

Après avoir examiné ces changements, Alice peut les merger dans sa branche "master":

```
$ git merge bob/master
```

Ce `merge` peut aussi être fait en récupérant les données depuis sa propre branche de suivi distant, comme ceci:

```
$ git pull . remotes/bob/master
```

Git récupère toujours les merges dans la branche courante, quelques soient les options de la ligne de commande.

Plus tard, Bob peut mettre à jour son dot avec les dernières modifications d'Alice en utilisant:

```
$ git pull
```

Il n'a besoin de donner le chemin vers le dot d'Alice; quand Bob a cloné le dépôt d'Alice, git a stocké l'adresse de son dépôt dans la configuration du dépôt, et cette adresse est utilisée pour récupérer les données avec 'pull':

```
$ git config --get remote.origin.url
/home/alice/project
```

(La configuration complète créée par git-clone est visible en lançant "git config -l", et la page de documentation de git config explique chacune de ces options.)

Git conserve aussi une copie propre de la branche "master" d'Alice, sous le nom "origin/master":

```
$ git branch -r
  origin/master
```

Si Bob décide plus tard de travailler avec un hébergeur différent, il pourra toujours créer des clones et récupérer les données en utilisant le protocole ssh:

```
$ git clone alice.org:/home/alice/project myrepo
```

D'une autre manière, git contient un protocole natif, ou peut aussi utiliser rsync ou http; voir git pull pour plus de détails.

Git peut aussi être utilisé de manière plus similaire à CVS, avec un dépôt central sur lequel de nombreux utilisateurs envoient leur modifications; voir git push et gitcvs-migration.

Les Dépôt Git Publiques

Une autre façon d'envoyer des modifications à un projet est d'avertir le chef de ce projet afin qu'il récupère les changements depuis votre dépôt en utilisant git pull. C'est un manière d'obtenir les mises à jours du dépôt principal, mais cela fonctionne aussi dans l'autre sens.

Si vous et le chef de projets avaient tous les 2 un compte sur le même ordinateur, alors vous pouvez échanger les modifications de vos dépôts respectifs directement; les commandes qui acceptent des URL de dépôts comme options, accepteront aussi un chemin de répertoire local:

```
$ git clone /path/to/repository
$ git pull /path/to/other/repository
```

ou une adresse ssh:

```
$ git clone ssh://yourhost/~you/repository
```

Pour les projets avec quelques développeurs, ou pour synchroniser quelques projets privés, cela peut vous suffire.

Cependant, la pratique la plus courante est de maintenir un dépôt publique (généralement sur le même host) pour que les autres puissent y récupérer les changements. Cela est souvent plus efficace, et vous permet de séparer proprement le travail privé en cours de réalisation des projets publiques et visibles.

Vous continuerez à travailler au jour-le-jour sur votre dépôt personnel, mais périodiquement vous enverrez (push) les modifications de votre dépôt personnel sur votre dépôt publique, permettant alors aux autres développeurs de récupérer (pull) les changements disponible dans ce dépôt. Donc le flux de travail, dans une situation où un autre développeur fournit des changement dans son dépôt publique, ressemble à ça:

```
                         vous envoyez (push)
votre dépôt personnel ---------------------------> votre dépot publique
          ^                                               |
          |                                               |
          | vous récupérez (pull)                         | ils récupèrent (pull)
          |                                               |
          |                                               |
          |               ils envoient (push)             V
leur dépôt publique <------------------------------  leur dépôt
```

Publier des Modifications sur un Dépôt Publique

L'utilisation des protocoles http et git permet aux autres développeurs de récupérer les derniers changements, mais ils n'auront pas l'autorisation d'écriture sur ces dépôt. Pour cela, vous devrez mettre à jour votre dépôt publique avec les derniers changements obtenus votre le dépôt privé.

La façon la plus simple de procéder est d'utiliser git push et ssh; pour mettre à jour la brancje "master" avec le dernier état de votre branche "master", lancez:

```
$ git push ssh://yourserver.com/~you/proj.git master:master
```

ou juste

```
$ git push ssh://yourserver.com/~you/proj.git master
```

Comem avec git-fetch, git-push se plaindra qu'il n'y a pas eu d'avance rapide (fast-forward); allez à la section suivante pour plus de détails pour gérer ce cas.

La cible d'un "push" est normalement un dépôt 'nu'. Vous pouvez aussi publier vers un dépôt qui contient une arborescence de travail, mais cette arborescence ne sera pas mise à jour durant la publication. Cela pourra vous amener à des résultats inattendus, par exemple si la branche que vous publiez est l'actuelle branche de travail sur le dépôt publique.

Comme avec git-fetch, vous pouvez aussi rajouter des options de configuration pour vous permettre d'aller plus rapidement; par exemple, après:

```
$ cat >>.git/config <<EOF
[remote "public-repo"]
    url = ssh://yourserver.com/~you/proj.git
EOF
```

vous devriez pouvoir effectuer les opérations précédentes avec juste

```
$ git push public-repo master
```

Voir les explications des options remote..url, branch..remote, et remote..push dans git config pour plus de détails.

Que faire quand une publication échoue

Si une publication ne se termine pas avec une avance de la branche distante, alors elle échouera avec un message d'erreur comme celui-ci:

```
error: remote 'refs/heads/master' is not an ancestor of
local  'refs/heads/master'.
Maybe you are not up-to-date and need to pull first?
error: failed to push to 'ssh://yourserver.com/~you/proj.git'
```

Cela peut arrivé, par exemple, si:

```
- vous avez utilisé `git-reset --hard` pour effacer un commit déjà publié, ou
- vous avez utilisé `git-commit --amend` pour remplacer un commit déjà publié, ou
- vous avez utilisé `git-rebase` pour recombiner un commit déjà publié.
```

Vous pouvez forcer un git-push à effectuer quand même la mise à jour en rajoutant le préfixe '+' au nom de la branche:

```
$ git push ssh://yourserver.com/~you/proj.git +master
```

Normalement, quand le sommet de la branche d'un dépôt publique est modifié, il est modifié pour pointer vers un descendant du commit ver lequel il pointait avant. En forçant la publication dans cette situation, vous cassez cette convention.

Néanmoins, c'est une bonne pratique pour les gens qui ont besoin de publier simplement une série de patches des travaux en cours, et c'est un compromis acceptable tant que vous prévenez les autres développeurs que vous comptez gérer la branche de cette façon.

Il est aussi possible qu'une publication échoue de cette façon quand d'autres personnes ont le droit de publier sur le même dépôt. Dans ce cas, la solution la plus correcte est de retenter la publication après avoir mis à jour votre travail: soit par un git-pull, soit par un git-fetch suivi d'une recombinaison (rebase); voir la prochaine partie et gitcvs-migration pour plus de détails.

TAG AVEC GIT

Les Tags Légers

Vous pouvez créer un tag qui référence un commit particulier de git en lançant git tag en spécifiant le nom du tag et le nom SHA1 du commit à tagger en options.

```
$ git tag stable-1 1b2e1d63ff
```

Après ça, nous pouvons utilisé stable-1 pou faire référence au commit 1b2e1d63ff.

Ceci est un "tag léger", car aucune branche n'a été créée durant le processus. Si vous voulez aussi rajouter un commentaire à ce tag, et le signer avec un méthode cryptographique, alors nous allons plutôt créé un *objet tag*.

Les Objets Tags

Si une de ces options: **-a**, **-s**, ou **-u <key-id>** est passé en argument de la commande, alors nous créons un objet tag qui nécessite un message tag. À moins que l'option -m ou -F soit fournie, un éditeur se lance pour que l'utilisateur saisisse le message du tag.

Quand cela se produit, un nouvelle objet est ajouté à la base de donnée objet de Git et la référence tag pointe vers cet *objet tag*, plutôt que sur le commit lui-même. Ce concept est utile car vous pouvez maintenant signer le tag, et vérifier plus tard que le commit correspond. Vous pouvez créer un objet tag comme ceci:

```
$ git tag -a stable-1 1b2e1d63ff
```

Il est possible de tagger n'importe quel objet, mais le taggage des objets "commit" est la pratique la plus courante. (Dans les sources du kernel Linux, le premier objet tag fait référence à un "tree", plutôt qu'un "commit").

Signature des Tags

Si vous avez configuré votre clé GPG, vous pouvez facilement créer des tags signés. Dans un premier temps, vous devrez configurer l'identifiant de votre clé dans votre fichier de configuration *.git/config* ou *~.gitconfig*.

```
[user]
    signingkey = <gpg-key-id>
```

Vous pouvez aussi le configurer comme ceci:

```
$ git config (--global) user.signingkey <gpg-key-id>
```

Maintenant vous pouvez créer un tag signé en remplaçant juste le **-a** par un **-s**.

```
$ git tag -s stable-1 1b2e1d63ff
```

Si vous n'avez pas configuré une clé GPG dans votre fichier de configuration, vous pouvez faire la même chose de cette façon:

```
$ git tag -u <gpg-key-id> stable-1 1b2e1d63ff
```

Chapter 4

Usage Intermédiaire

IGNORER DES FICHIERS

Un projet générera souvent des fichiers que vous ne voulez pas surveiller avec git. En général, ceux sont des fichiers qui ne servent que compilation de programme, ou des fichiers temporaires créés par votre éditeur de texte. Ne pas surveiller ces fichiers revient à ne pas les inclure avec "git-add". Mais il devient très vite ennuyant d'avoir tous ces fichiers non-suivis; ils rendent "git add ." et "git commit -a" pratiquement inutile, et ils se montrent dans chaque sortie de "git status".

Vous pouvais demander à git d'ignorer certains fichiers en créant un fichier nommé .gitignore dans la racine de votre répertoire de travail. Ce fichier contiendra ce type d'information:

```
# Les ligne commençant par '#' sont des commentaires.
# Ignorer tous les fichiers nommés foo.txt
foo.txt
# Ignorer tous les fichiers html
```

```
*.html
# à l'exception de foo.html qui est maintenu à la main
!foo.html
# Ignorer les objets et les archives
*.[oa]
```

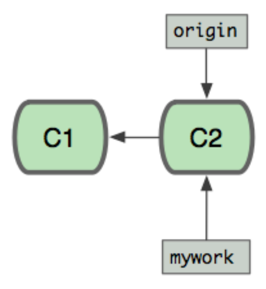
Voir gitignore pour une explication plus détaillée de la syntaxe. Vous pouvez aussi placer .gitignore dans un autre sous-répertoire de votre répertoire de travail, ses règles s'appliqueront alors seulement au répertoire et sous-répertoire de là où il se trouve. Le fichier `.gitignore` peut être ajouté au dépôt comme n'importe quel autre fichier (lancez juste `git add .gitignore` and `git commit`, comme d'habitude), cela est utile quand les chaînes d'exclusion (comme celle pour exclure les fichiers de compilation) peuvent aussi être utiles aux autres développeurs clonant votre dépôt.

Si vous voulez que la chaîne d'exclusion n'affecte que certains sous-répertoires (plutôt que tout le dépôt pour un certain projet), vous devrez alors peut-être mettre cette chaîne dans un fichier de votre dépôt nommé .git/info/exclude, ou dans un des fichiers spécifiés par la variable de configuration `core.excludesfile`. Certaines commandes git peuvent aussi prendre ces chaînes d'exclusion directement comme argument de la ligne de commande. Voir gitignore pour plus de détails.

RECOMBINAISON (REBASE)

Supposons que vous avez créé une branche "mywork" sur une branche de suivi distante "origin".

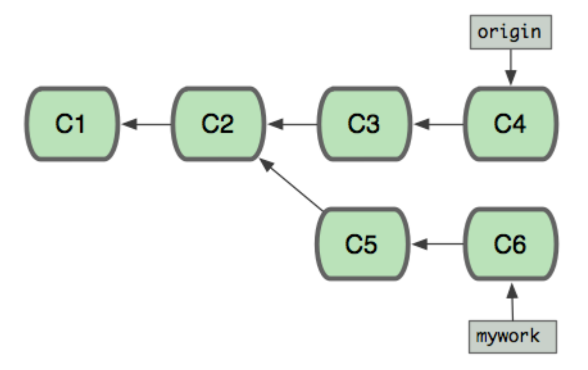
```
$ git checkout -b mywork origin
```



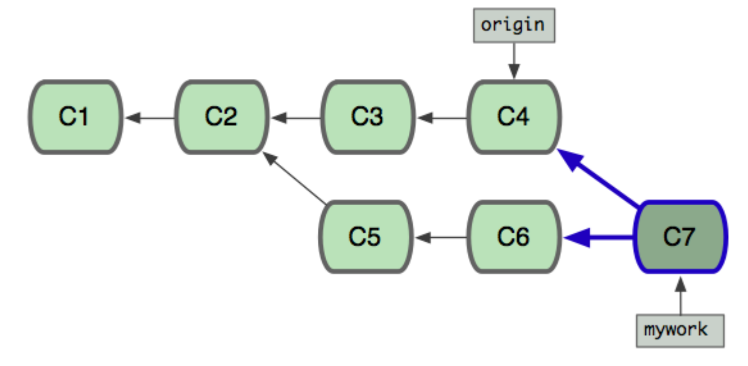
Maintenant travaillez un peu dessus, en créant 2 commits.

```
$ vi fichier.txt
$ git commit
$ vi autrefichier.txt
$ git commit
...
```

Pendant ce temps, quelqu'un d'autre travaille en créant aussi 2 nouveaux commits sur la branche d'origine. Cela signifie que les 2 branches 'origine' et 'mywork' ont avancées, et elle ont aussi divergées.



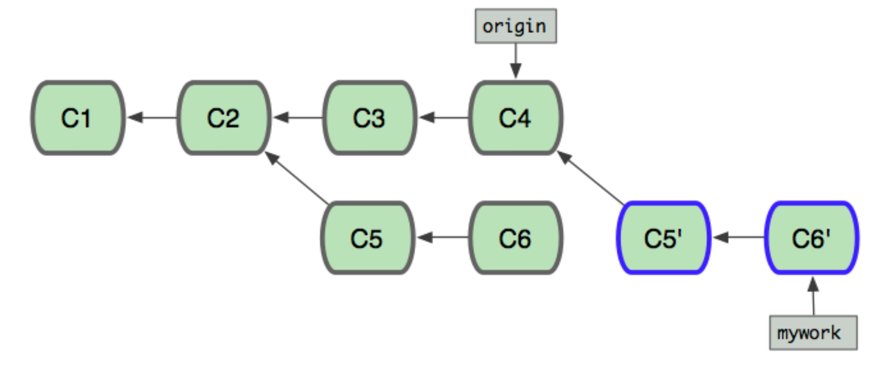
À ce moment, vous pouvez utiliser "pull" pour merger vos modifications; le résultat créera un nouveau commit 'merge', comme ceci:



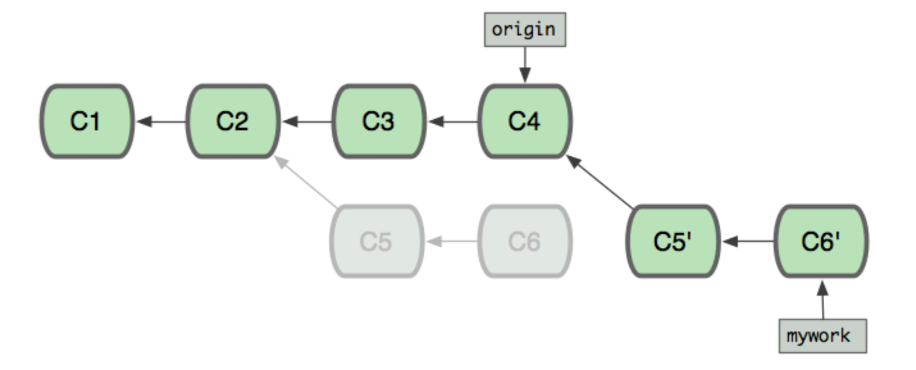
Cependant, si vous préférez garder l'historique de 'mywork' sous l'aspect d'une simple série de commits sans 'merge', vous pouvez aussi utiliser git rebase:

```
$ git checkout mywork
$ git rebase origin
```

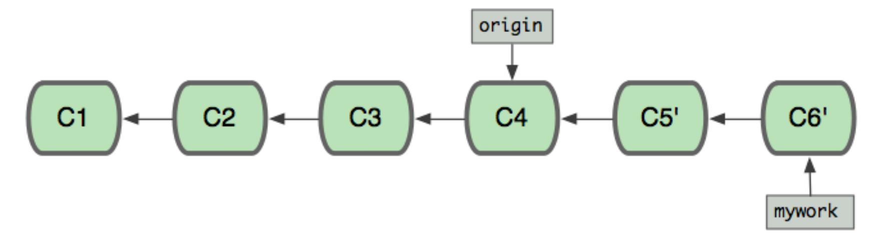
Cette commande va retirer chacun de vos commit sur 'mywork', en les sauvegardant temporairement comme des patches (dans le dossier ".git/rebase"), puis mettre à jour la branche 'mywork' avec la dernière version de la branche 'origin', et enfin appliquer chaque patch sauvegardé à cette nouvelle version de 'mywork'.

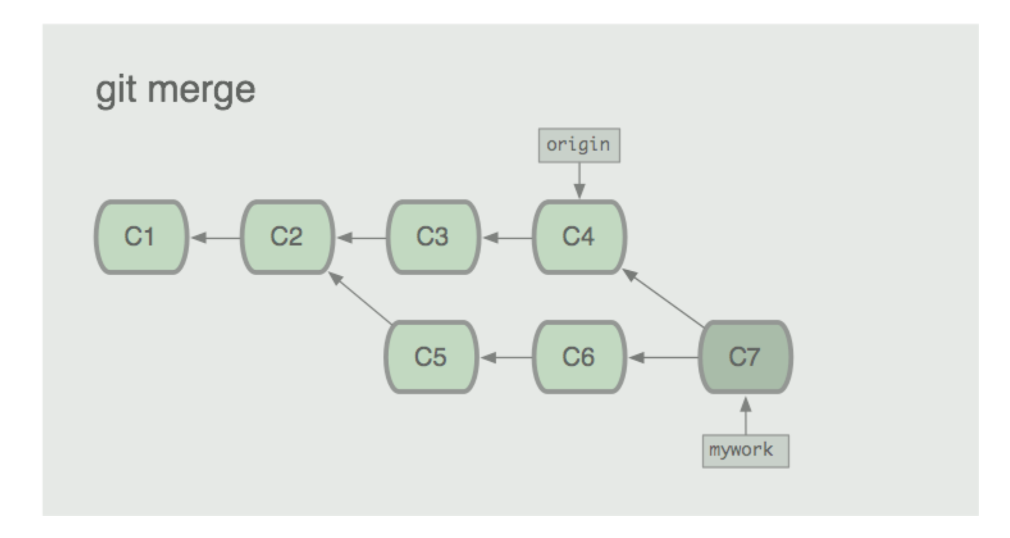


Une fois que la référence ('mywork') est mise à jour jusqu'au dernier objet commit créé, vos anciens commits seront abandonnés. Ils seront sûrement effacer si vous lancer la commande de ramasse-miettes. (voir git gc)



Maintenant nous pouvons voir la différence de l'historique entre l'exécution d'un merge et l'exécution d'une recombinaison (rebase):


git rebase
origin
C1
C2
C3
C4
C5'
C6'
mywork


git merge
origin
C1
C2
C3
C4
C5
C6
C7
mywork

Dans le processus d'une recombinaison, des conflits peuvent se produire. Dans ce cas, le processus s'arrêtera et vous permettra de réparer ces conflits; après les avoir fixés, utilisez "git-add" pour mettre à jour l'index avec ce nouveau contenu, puis, au lieu de lancer git-commit, lancez juste:

```
$ git rebase --continue
```

et git continuera d'appliquer le reste des patches.

À n'importe quel moment, vous pouvez utiliser l'option `--abort` pour annuler le processus et retourner au même état de 'mywork' qu'au démarrage de la recombinaison:

```
$ git rebase --abort
```

RECOMBINAISON INTERACTIVE

Vous pouvez aussi recombiner (rebase) les commits interactivement. Ceci s'utilise souvent pour ré-écrire vos objets commits avant de les publier. C'est une manière simple de découper, regrouper et ré-ordonner les commits avant de les partager avec les autres. Vous pouvez aussi utiliser ça pour nettoyer les commits que vous récupérerez chez quelqu'un avant de les appliquer localement.

Si vous voulez modifier interactivement vos commit durant la recombinaison, vous pouvez activer le mode interactif en utilisant l'option '-i' ou '--interactive' avec la commande 'git rebase'.

```
$ git rebase -i origin/master
```

Cela lancera le mode interactif de recombinaison, avec tous les commits que vous avez créé depuis votre dernière publication (ou le dernier merge depuis le dépôt d'origine).

Pour voir quels commits seront concernés, vous pouvez utiliser la commande 'log' de cette façon:

```
$ git log github/master..
```

Quand vous lancerez la commande 'rebase -i', vous vous trouverez dans un éditeur qui ressemblera à ça:

```
pick fc62e55 added file_size
pick 9824bf4 fixed little thing
pick 21d80a5 added number to log
pick 76b9da6 added the apply command
pick c264051 Revert "added file_size" - not implemented correctly

# Rebase f408319..b04dc3d onto f408319
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Cela signifie qu'il y a 5 commits depuis votre dernière publication et chaque commit est décrit par un ligne avec le format suivant:

```
(action) (sha partiel) (court message du commit)
```

Maintenant, vous pouvez changer l'action (qui est 'pick' par défaut) soit par 'edit', ou 'squash', ou juste la laisser comme elle est ('pick'). Vous pouvez aussi ré-ordonner les commits en déplaçant les lignes comme vous le voulez. Ensuite, quand vous sortez de l'éditeur, git essayera d'appliquer les commits en suivant leur ordre d'arrangement et l'action sélectionnée.

Si 'pick' est sélectionné, il essayera simplement d'appliquer le patch et de sauvegarder le commit avec le même message qu'avant.

Si 'squash' est sélectionné, il combinera ce commit avec le précédent pour former un nouveau commit. Vous trouverez alors un autre éditeur pour merger les messages des 2 commit qui ont étés assemblés ensemble. Donc, si vous sortez du premier éditeur de la manière suivante:

```
pick   fc62e55 added file_size
squash 9824bf4 fixed little thing
squash 21d80a5 added number to log
squash 76b9da6 added the apply command
squash c264051 Revert "added file_size" - not implemented correctly
```

vous devrez créer un nouveau message de commit `partir de ça`:

```
# This is a combination of 5 commits.
# The first commit's message is:
added file_size

# This is the 2nd commit message:

fixed little thing

# This is the 3rd commit message:

added number to log

# This is the 4th commit message:

added the apply command

# This is the 5th commit message:
```

```
Revert "added file_size" - not implemented correctly

This reverts commit fc62e5543b195f18391886b9f663d5a7eca38e84.
```

Une fois que vous aurez édité cette partie en un seul message et quitté l'éditeur, le commit sera sauvegardé avec votre nouveau message.

Si vous avec sélectionné 'edit', la même chose se passera, mais un pause sera marqué entre chaque commit pour vous donner la main avec une ligne de commande afin que vous puissiez modifier le commit, ou son contenu.

Par exemple, si vous voulez découper un commit, vous sélectionnerez 'edit' pour ce commit:

```
pick   fc62e55 added file_size
pick   9824bf4 fixed little thing
edit   21d80a5 added number to log
pick   76b9da6 added the apply command
pick   c264051 Revert "added file_size" - not implemented correctly
```

RY quand vous vous trouverez avec la ligne de commande, vous pourrez revenir sur ce commit pour en créer deux (ou plus) nouveaux. Disons que 21d80a5 modifie 2 fichiers, fichier1 et fichier2, et que vous voulez le découper en 2 commits séparés. Vous pouvez faire ceci quand la recombinaison vous redonne la main avec la ligne de commande:

```
$ git reset HEAD^
$ git add fichier1
$ git commit 'première partie du commit découpé'
$ git add fichier2
$ git commit 'seconde partie du commit découpé'
$ git rebase --continue
```

Et maintenant, au lieu d'avoir 5 commits, vous en avez 6.

La recombinaison interactive peut vous aider sur un dernier point utilise, elle peut oublier des commits. Si au lieu de sélectionner 'pick', 'squash' ou 'edit' pour la ligne de commit, vous effacez simplement la ligne, alors le commit sera retiré de l'historique.

AJOUT INTERACTIF

Il est vraiment intéressant de travailler et de visualiser l'index de Git avec les ajouts interactifs. Pour le démarrer, tape simplement 'git add -i'. Git vous montrera tous vos fichiers modifiés et leur état.

```
$>git add -i
           staged     unstaged path
  1:    unchanged        +4/-0 assets/stylesheets/style.css
  2:    unchanged      +23/-11 layout/book_index_template.html
  3:    unchanged        +7/-7 layout/chapter_template.html
  4:    unchanged        +3/-3 script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown

*** Commands ***
  1: status   2: update   3: revert   4: add untracked
  5: patch    6: diff     7: quit     8: help
What now>
```

Dans ce cas, nous pouvons voir 5 fichiers modifiés qui n'ont pas encore étés ajoutés à l'index (unstaged), et le nombre de lignes en plus ou en moins pour chacun d'eux. Nous trouvons aussi un menu interactif avec les possibilités de cet outils.

Si nous voulons rajouter les fichiers dans l'index, nous pouvons taper '2' ou 'u' pour le mode de mise à jour (update). Après, nous devons sélectionner les fichiers que nous voulons rajouter à l'index en saisissant les numéros de ces fichiers (dans ce cas, 1-4):

```
What now> 2
           staged     unstaged path
  1:    unchanged        +4/-0 assets/stylesheets/style.css
  2:    unchanged      +23/-11 layout/book_index_template.html
  3:    unchanged        +7/-7 layout/chapter_template.html
  4:    unchanged        +3/-3 script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
Update>> 1-4
           staged     unstaged path
* 1:    unchanged        +4/-0 assets/stylesheets/style.css
* 2:    unchanged      +23/-11 layout/book_index_template.html
* 3:    unchanged        +7/-7 layout/chapter_template.html
* 4:    unchanged        +3/-3 script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
Update>>
```

Si je presse 'entrée', je reviendrai au menu principal où je peux voir que l'état des fichiers a changé.

```
What now> status
           staged     unstaged path
  1:        +4/-0      nothing assets/stylesheets/style.css
  2:      +23/-11      nothing layout/book_index_template.html
  3:        +7/-7      nothing layout/chapter_template.html
  4:        +3/-3      nothing script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
```

Maintenant nous voyons que les 4 premiers fichiers sont assemblés et que le dernier ne l'est pas encore. Ces informations sont simplement une compression de l'affichage obtenu avec la command 'git status':

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
#   modified:   assets/stylesheets/style.css
#   modified:   layout/book_index_template.html
#   modified:   layout/chapter_template.html
#   modified:   script/pdf.rb
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
#
```

Il y a quelques options utiles dans ce mode d'ajout interactif, comme retirer des fichiers de l'index (3: revert), ajouter des fichier non-suivis (4: add untracked), et voir les différences (6: view diff). Ces options sont assez faciles à comprendre. Cependant, il y a une commande plutôt cool qui demande des explications: l'assemblage de patches (5: patch).

Si vous tapez '5' ou 'p' dans le menu, git vous montrera vos différences patch par patch (ou morceau par morceau) et vous demandera si vous voulez assembler chacun d'eux. De cette façon, vous pouvez n'assembler qu'une partie d'un fichier modifié. Si vous avez édité un fichier et ne voulez committer qu'une partie des modifications et laisser les autres modifications dans un état inachevé, ou séparer les commits de documentation de ceux des changements plus conséquents, vous pouvez utiliser 'git add -i' pour le faire facilement.

Ici j'ai assemblé quelques changements au fichier book_index_template.html, mais pas tous:

```
           staged     unstaged path
  1:        +4/-0      nothing assets/stylesheets/style.css
  2:       +20/-7        +3/-4 layout/book_index_template.html
  3:        +7/-7      nothing layout/chapter_template.html
  4:        +3/-3      nothing script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
  6:    unchanged       +85/-0 text/15_Interactive_Adding/0_ Interactive_Adding.markdown
```

Quand vous avez terminer de modifier votre index avec 'git add -i', il vous suffit de quitter (7: quit) et de lancer 'git commit' pour committer les changements assemblés. Souvenez-vous de **ne pas lancer** 'git commit -a', cela effacerait tous les précieuses sélections que vous avez faites dans le mode interactif et committera tout d'un coup.

LE CACHE (STASHING)

Quand vous êtes au milieu d'une grosse modification, il se peut que vous trouviez un énorme bug qui doit être corrigé rapidement. Vous aimeriez le corriger avant de continuer vos modifications. Vous pouvez utiliser git stash pour sauvegarder l'état actuel de votre travail, et une fois le bug éliminé (ou, éventuellement, en revenant sur la branche après l'avoir corrigé sur une branche différente), retrouver les changement du travail en cours.

```
$ git stash "travail en cours de la fonctionnalité conquerir-le-monde"
```

Cette commande sauvegardera vos changement dans le `stash`, et remettra à zéro votre répertoire de travail et l'index afin de correspondre avec le sommet de la branche courante. Après vous pouvez faire vos corrections comme d'habitude.

```
... éditer et tester ...
$ git commit -a -m "machin: bidulefix"
```

Après ça, vous pouvez revenir là où vous étiez en train de travailler avec `git stash apply`:

```
$ git stash apply
```

La File de Cache (Stash Queue)

Vous pouvez aussi utiliser le cache pour empiler vos changements cachés. Si vous lancez 'git stash list', vous verrez quelles caches vous avez sauvegardé:

```
$>git stash list
stash@{0}: WIP on book: 51bea1d... fixed images
stash@{1}: WIP on master: 9705ae6... changed the browse code to the official repo
```

Ensuite vous pouvez les sélectionner individuellement pour les appliquer avec 'git stash apply stash@{1}'. Vous pouvez nettoyer la liste avec 'git stash clear'.

L'ARBORESCENCE GIT (TREEISH)

Il y a de nombreuses manières de faire référence à un "commit" ou un "tree" sans épeler toute la chaîne de 40 caractères sha. Avec Git, on appelle ces références 'treeish'.

Sha Partiel

Si le sha de votre commit est '`980e3ccdaac54a0d4de358f3fe5d718027d96aae`', git le reconnaîtra aussi avec les chaîne suivante:

```
980e3ccdaac54a0d4de358f3fe5d718027d96aae
980e3ccdaac54a0d4
980e3cc
```

Tant que le sha partiel est unique - il ne peut pas être confondu avec un autre (ce qui serait vraiment invraisemblable si vous avez au moins 5 caractères), git étendra le sha partiel pour vous.

Branche, Remote ou Nom de Tag

Vous pouvez toujours utiliser une branche, un remote ou un nom de tag à la place d'un sha, puisque qu'ils ne sont que des pointeurs. Si votre branche "master" est sur le commit 980e3 et que vous la publiez (push) sur 'origin' et que vous l'avez taggée 'v1.0', alors toutes les références suivantes sont équivalentes:

```
980e3ccdaac54a0d4de358f3fe5d718027d96aae
origin/master
refs/remotes/origin/master
master
refs/heads/master
v1.0
refs/tags/v1.0
```

Ce qui signifie que les commandes suivantes vous donneront le même résultat:

```
$ git log master

$ git log refs/tags/v1.0
```

Spécification de Date

Le Log de Références que git conserve vous permettre de faire certaines choses localement, comme:

```
master@{yesterday}

master@{1 month ago}
```

Qui est un raccourci pour 'là où se trouver le sommet de la branche master hier', etc. Ce format peut montrer des sha différents sur des ordinateurs différents, même si la branche master est actuellement au même niveau.

Spécification Ordinale

Le format suivant vous donnera la Nième valeur précédent une référence particulière. Par exemple:

```
master@{5}
```

vous donnera la 5ième valeur avant la référence du sommet de master.

La Carotte Parent

Cela vous donnera le Nième parent d'un commit particulier. Ce format n'est utile que pour les commits de merge - les objets commits qui ont plus d'un parent.

```
master^2
```

Spécification avec le Tilde

La spécification avec le tilde vous donnera le Nième grand-parent d'un objet commit. Par exemple,

```
master~2
```

vous donnera le premier parent du premier parent du commit vers lequel pointe master. C'est l'équivalent de:

```
master^^
```

Vous pouvez aussi continuer à faire ça. Les spécifications suivantes pointent vers le même commit:

```
master^^^^^^
master~3^~2
master~6
```

Pointeur de Tree

Cela facilite la recherche d'un commit en partant de l'arbre vers lequel il pointe. Si vous avez besoin du sha vers lequel un commit pointe, vous pouvez ajouter la spécification '^{tree}' à la fin de celui-ci.

```
master^{tree}
```

Spécification de Blob

Si vous voulez le sha d'un blob en particulier, vous pouvez ajouter le chemin du blob à la fin de l'arborescence, comme ceci:

```
master:/chemin/vers/le/fichier
```

Les Suites

Pour terminer, vous pouvez spécifier une suite de commits avec la spécification de suite. Par exemple, vous pourrez obtenir les commits entre 7b593b5 et 51bea1 (avec 51bea1 le plus récent), en éliminant 7b593b5 mais en gardant 51bea1:

```
7b593b5..51bea1
```

Ceci inclura tous les commits *depuis* 7b593b:

```
7b593b..
```

LES BRANCHES DE SUIVI

Une 'branche de suivi' de Git est une branche locale qui est connectée à une branche distante. Quand vous publiez ou récupérez les données de cette branche, Git publie et récupères automatiquement les informations de la branche à laquelle elle est connectée.

Utilisez ceci si vous récupérer toujours vos données depuis la même branche en amont dans une nouvelle branche, et si vous ne voulez pas utiliser "git pull " explicitement.

La commande 'git clone' configure automatiquement une branche "master" qui est une branche de suivi pour 'origin/master' - la branche "master" du dépôt cloné.

Vus pouvez créer un branche de suivi manuellement en ajoutant l'option '--track' à la commande 'branch' de Git.

```
git branch --track experimental origin/experimental
```

Puis vous lancez::

```
$ git pull experimental
```

Cela récupérera automatiquement les données de 'origin' et mergera 'origin/experimental' dans votre branche local 'experimental'.

De a même manière, vous pouvez publier vers 'origin', Git publiera vos modifications de 'experimental' vers 'origin/experimental', sans n'avoir aucune commande à spécifier.

RECHERCHE AVEC GIT GREP

Il est très facile de trouver des fichiers avec des mots ou des phrases avec la commande git grep. Il est aussi possible de le faire avec la commande 'grep' unix, mais vous pouvez faire des recherches sur vos version précédentes avec 'git grep' sans avoir à les rapatrier.

Par exemple, si je veux voir tous les endroits qui appellent 'xmmap' dans mon dépôt git.git, je pourrai lancer ça:

```
$ git grep xmmap
config.c:               contents = xmmap(NULL, contents_sz, PROT_READ,
diff.c:         s->data = xmmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd, 0);
git-compat-util.h:extern void *xmmap(void *start, size_t length, int prot, int fla
read-cache.c:   mmap = xmmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_PRIVATE,
refs.c: log_mapped = xmmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, logfd, 0);
sha1_file.c:    map = xmmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, fd, 0);
sha1_file.c:    idx_map = xmmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd, 0);
sha1_file.c:                    win->base = xmmap(NULL, win->len,
sha1_file.c:                    map = xmmap(NULL, *size, PROT_READ, MAP_PRIVATE, f
sha1_file.c:            buf = xmmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
wrapper.c:void *xmmap(void *start, size_t length,
```

Si je veux aussi voir le numéro de ligne de chacun de ces résultats, je peux ajouter l'option '-n':

```
$>git grep -n xmmap
config.c:1016:          contents = xmmap(NULL, contents_sz, PROT_READ,
diff.c:1833:            s->data = xmmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd,
git-compat-util.h:291:extern void *xmmap(void *start, size_t length, int prot, int
read-cache.c:1178:      mmap = xmmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_
refs.c:1345:    log_mapped = xmmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, logfd, 0);
sha1_file.c:377:        map = xmmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, fd, 0);
sha1_file.c:479:        idx_map = xmmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd
sha1_file.c:780:                        win->base = xmmap(NULL, win->len,
```

```
sha1_file.c:1076:                       map = xmmap(NULL, *size, PROT_READ, MAP_PR
sha1_file.c:2393:               buf = xmmap(NULL, size, PROT_READ, MAP_PRIVATE, fd
wrapper.c:89:void *xmmap(void *start, size_t length,
```

Si nous sommes seulement intéressé par le nom du fichier, nous pouvons utiliser l'option '--name-only':

```
$>git grep --name-only xmmap
config.c
diff.c
git-compat-util.h
read-cache.c
refs.c
sha1_file.c
wrapper.c
```

Nous pouvons aussi voir combien lignes correspondent dans chaque fichier avec l'option '-c':

```
$>git grep -c xmmap
config.c:1
diff.c:1
git-compat-util.h:1
read-cache.c:1
refs.c:1
sha1_file.c:5
wrapper.c:1
```

Maintenant, si je veux voir dans quelle version de git ce code a été utilisé, je pourrai ajouter la référence du tag à la fin, comme ceci:

```
$ git grep xmmap v1.5.0
v1.5.0:config.c:                contents = xmmap(NULL, st.st_size, PROT_READ,
v1.5.0:diff.c:          s->data = xmmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:git-compat-util.h:static inline void *xmmap(void *start, size_t length,
```

```
v1.5.0:read-cache.c:                    cache_mmap = xmmap(NULL, cache_mmap_size,
v1.5.0:refs.c:  log_mapped = xmmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, logfd
v1.5.0:sha1_file.c:     map = xmmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:sha1_file.c:     idx_map = xmmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd
v1.5.0:sha1_file.c:                     win->base = xmmap(NULL, win->len,
v1.5.0:sha1_file.c:     map = xmmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:sha1_file.c:             buf = xmmap(NULL, size, PROT_READ, MAP_PRIVATE, fd
```

Nous pouvons voir les différences entre le code en cours et la version 1.5.0. Dans l'une d'elles, xmmap est maintenant utilisé dans wrapper.c alors qu'il ne l'était pas sans la v1.5.0.

Vous pouvez aussi combiner les termes de recherche dans grep. Par exemple, si nous cherchons où est définie SORT_DIRENT dans notre dépôt:

```
$ git grep -e '#define' --and -e SORT_DIRENT
builtin-fsck.c:#define SORT_DIRENT 0
builtin-fsck.c:#define SORT_DIRENT 1
```

Vous pouvez aussi rechercher tous les fichiers qui contiennent la *totalité* des termes recherchés, et affiche les lignes *avec l'un de* ces 2 termes:

```
$ git grep --all-match -e '#define' -e SORT_DIRENT
builtin-fsck.c:#define REACHABLE 0x0001
builtin-fsck.c:#define SEEN      0x0002
builtin-fsck.c:#define ERROR_OBJECT 01
builtin-fsck.c:#define ERROR_REACHABLE 02
builtin-fsck.c:#define SORT_DIRENT 0
builtin-fsck.c:#define DIRENT_SORT_HINT(de) 0
builtin-fsck.c:#define SORT_DIRENT 1
builtin-fsck.c:#define DIRENT_SORT_HINT(de) ((de)->d_ino)
builtin-fsck.c:#define MAX_SHA1_ENTRIES (1024)
builtin-fsck.c: if (SORT_DIRENT)
```

Vous pouvez aussi rechercher les lignes qui ont un terme et chacun de ces 2 termes, par exemple, si vous voulez voir où sont définies les constantes qui ont soit PATH ou MAX dans leur nom:

```
$ git grep -e '#define' --and \( -e PATH -e MAX \)
abspath.c:#define MAXDEPTH 5
builtin-blame.c:#define MORE_THAN_ONE_PATH      (1u<<13)
builtin-blame.c:#define MAXSG 16
builtin-describe.c:#define MAX_TAGS     (FLAG_BITS - 1)
builtin-fetch-pack.c:#define MAX_IN_VAIN 256
builtin-fsck.c:#define MAX_SHA1_ENTRIES (1024)
...
```

RÉPARATION AVEC GIT - RESET, CHECKOUT ET REVERT

Git fournit plusieurs méthodes pour réparer vos erreurs durant votre développement. Choisir la bonne méthode dépend si si vous avez committé ou non votre erreur, et si elle a été committée, si cette erreur a déjà été récupérée par un autre développeur.

Réparer une erreur non-committée

Si vous vous Êtres embrouillez dans votre répertoire de travail, mais que vous n'avez pas encore committé vos erreurs, vous pouvez retrouver l'intégrité de votre répertoire dans l'état qu'il était après le dernier commit en utilisant:

```
$ git reset --hard HEAD
```

Cela effacera toutes les modifications que vous avec ajouté à l'index git et aussi les changements qui sont présents dans votre répertoire de travail mais qui n'ont pas été ajoutés à l'index. En d'autres mots, après cette commande, le résultat de "git diff" et "git diff --cached" sera vide.

Si vous ne voulez restaurer qu'un seul fichier, par exemple hello.rb, utiliser plutôt git checkout

```
$ git checkout -- hello.rb
$ git checkout HEAD hello.rb
```

La première commande restaure hello.rb à la version de l'index, afin que "git diff hello.rb" ne retourne aucune différence. La seconde commande restaurera hello.rb à la version de la révision HEAD, afin que "git diff hello.rb" et "git diff --cached hello.rb" ne retourne aucune différence.

Réparer une erreurs committée

Si vous avez déjà committé ce que vous n'auriez pas du, il y a 2 façons fondamentalement différentes de régler le problème:

1. vous pouvez créer un nouveau commit qui annule les changements du dernier commit. C'est la manière correcte de s'y prendre si votre erreur est déjà publique.

2. Vous pouvez revenir en arrière et modifier l'ancien commit. Vous ne devriez jamais faire ça si vous avez déjà rendu l'historique publique; git n'est pas conçu pour que l'historique d'un projet change, et ne peut pas effectuer correctement des merges répétés sur depuis branche qui a vu son historique changer.

Réparer une erreur sur un nouveau commit

Il est facile de créer un nouveau commit qui annule les changements d'un commit précédent; utilisez la commande git revert avec la référence du mauvais commit; par exemple, pour revenir au commit le plus récent:

```
$ git revert HEAD
```

Cela créera un nouveau commit qui annule les changements dans HEAD. Vous pourrez éditer le message de ce nouveau commit.

Vous pouvez aussi revenir sur des changements plus anciens, par exemple, sur l'avant-dernier changement:

```
$ git revert HEAD^
```

Dans ce cas, git essayera d'annuler l'ancien changement en gardant intactes les modifications faites depuis. Si plus d'un changement se superpose sur les changements à annuler, vous aurez à régler les conflits manuellement, de la même façon que quand vous réglez un merge.

Réparer une erreur en modifiant le commit

Si vous venez de committer quelque chose mais que vous vous rendez compte que vous devez réparer ce commit, les versions récentes de git commit vous donnent accès à l'option **--amend** qui demande à git de remplacer le commit de HEAD par un autre, basé sur le contenu actuel de l'index. Cela vous donne l'opportunité d'ajouter de fichiers que vous avez oubliés ou de corriger des erreurs de typo dans le message du commit, avant de publier les changements pour les autre développeurs.

Si vous trouvez une erreur dans un ancien commit, mais que vous ne l'avez toujours pas publié, utilisez le mode interactif de git rebase, avec "git rebase -i" en marquant les changements qui doivent être corrigés avec **edit**. Cela vous permettra de modifier le commit pendant le processus de recombinaison.

MAINTENIR GIT

Assurer une bonne performance

Sur les dépôts de grande taille, git nécessite une compression pour éviter que les informations de l'historique prennent trop d'espace sur le disque et en mémoire.

Cette compression n'est pas effectuée automatiquement. Vous devrez donc, à l'occasion, lancer git gc:

```
$ git gc
```

pour re-compresser l'archive. Cela peut prendre beaucoup de temps, donc vous préférerez peut-être lancer git-gc quand vous ne travaillez pas sur autre chose.

Assurer la fiabilité

La commande git fsck lance une série de vérification de consistance du dépôt, et rapporte les problèmes. Cela peut aussi prendre un peu de temps. L'avertissement le plus courant concerne les objets "dangling" (suspendus):

```
$ git fsck
dangling commit 7281251ddd2a61e38657c827739c57015671a6b3
dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63
dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5
dangling blob 218761f9d90712d37a9c5e36f406f92202db07eb
dangling commit bf093535a34a4d35731aa2bd90fe6b176302f14f
dangling commit 8e4bec7f2ddaa268bef999853c25755452100f8e
dangling tree d50bb86186bf27b681d25af89d3b5b68382e4085
dangling tree b24c2473f1fd3d91352a624795be026d64c8841f
...
```

Les objets suspendus ne posent aucun problèmes. Au pire ils prennent un petit peu plus d'espace disque. Ils servent parfois de solution de dernier secours pour récupérer des travaux perdus.

CONFIGURER UN DÉPÔT PUBLIAUE

En supposant que vous avez votre dépôt personnel dans le dossier ~/proj, nous allons créer un nouveau clone de ce dépôt et prévenir git-daemon qu'il est destiné à être publique:

```
$ git clone --bare ~/proj proj.git
$ touch proj.git/git-daemon-export-ok
```

Le dossier proj.git obtenu contient un dépôt git "nu" (bare)--il contient juste le contenu du répertoire ".git", sans aucun autres fichiers récupérés autour de ce dossier.

Ensuite, copiez proj.git sur le serveur où vous allez héberger votre dépôt publique. Vous pouvez utiliser scp, rsync, ou ce qui vous convient le plus.

Exporter un dépôt git avec le protocole git

C'est la méthode la plus utilisée.

Si quelqu'un d'autre s'occupe de l'administration de votre serveur, il devrait vous indiquer dans quel répertoire mettre votre dépôt, et sur quelle adresse git:// vous pourrez le récupérer.

Sinon, vous devez démarrer git daemon; il ecoutera le port 9418. Par défaut, il autorise les accès à tout ce qui ressemble à un répertoire git, et qui contient le fichier magique git-daemon-export-ok. Si vous ajoutez le chemin de quelques dossiers en options de git-daemon, il restreindra les exportation en ne tenant compte que de ces dossiers.

Vous pouvez aussi lancer git-daemon comme service inetd; regardez le manuel de git daemon; (man git-daemon), particulièrement les exemples, pour plus d'informations.

Exporter un dépôt git par http

Le protocole git est plus performant et plus fiable, mais sur un hébergeur qui a déjà configuré son serveur web, l'exportation http peut être une solution plus simple à mettre en place.

Tout ce que vous devez faire est de placer le dépôt git (bare) que vous venez de créer dans un dossier qui est accessible depuis le serveur web, et faire quelques ajustements pour donner au client web les quelques informations additionnelles qu'il a besoin:

```
$ mv proj.git /home/you/public_html/proj.git
$ cd proj.git
$ git --bare update-server-info
$ chmod a+x hooks/post-update
```

Pour plus d'explications sur ces 2 dernières lignes, regardez la documentation de git update-server-info et githooks.

Il vous suffit ensuite de diffuser l'adresse de proj.git aux autres développeurs. Tout le monde aura la possibilité de cloner ou de récupérer (pull) les changements du projet depuis cette adresse, avec la commande suivante par exemple:

```
$ git clone http://yourserver.com/~you/proj.git
```

CONFIGURER UN DÉPÔT PRIVÉ

Si vous devez configurer un dépôt privé, et que vous désirez faire ça localement, plutôt que d'utiliser un hébergeur, vous avez plusieurs solutions.

Accéder au Dépôt par SSH

En général, la solution la plus facile est d'utiliser simplement Git par SSH. Si les utilisateurs ont déjà des comptes ssh sur le serveur, vous pouvez placer le dépôt git n'importe où dans le système auquel ils ont accès, et les laisser accéder à ce dépôt grâce à une simple connexion ssh. Par exemple, disons que vous avez un dépôt que vous voulez héberger. Vous pouvez l'exporter comme un dépôt 'nu' puis le copier sur votre serveur avec 'scp' comme ceci:

```
$ git clone --bare /home/user/mondepot/.git /tmp/mondepot.git
$ scp -r /tmp/mondepot.git monserveur.com:/opt/git/mondepot.git
```

Puis, quelqu'un d'autre avec un compte ssh sur monserveur.com peut cloner votre dépôt via:

```
$ git clone myserver.com:/opt/git/myrepo.git
```

Cette commande leur demandera seulement leur mot de passe ssh ou leur clé publique, suivant la configuration de l'authentification ssh.

Accès multi-utilisateurs en utilisant Gitosis

Si vous ne voulez pas configurer des comptes différents pour chaque utilisateur, vous pouvez utiliser un outils nommé Gitosis. Dans gitosis, il y a un fichier authorized_keys qui contient les clés publiques de toutes les personnes autorisées à

accéder au dépôt, ensuite tout le monde peut utiliser l'utilisateur 'git' pour publier et récupérer les modifications sur le dépôt.

Poour plus d'informations sur Gitosis: Installing and Setting up Gitosis

Chapter 5

Git Avancé

CRÉATION DE NOUVELLES BRANCHES VIDES

Parfois, vous voudrez peut-être garder certaines branches dans votre dépôt qui ne partagent pas d'ancêtre en commun avec votre code normal. Des exemples de ce genre d'utilisation peut être de la documentation automatiquement générée ou des choses dans ce style. Si vous voulez créer une nouvelle branche qui n'utilise pas la base du code actuelle comme parent, vous pouvez créer une branche vide comme ceci:

```
git symbolic-ref HEAD refs/heads/nouvellebranche
rm .git/index
git clean -fdx
<travailler>
git add vos fichiers
git commit -m 'Premier commit'
```

MODIFIER VOTRE HISTORIQUE

La recombinaison interactive est une manière simple pour modifier des commits individuels.

La commande git filter-branch est une bonne façon de faire des commits en masse.

BRANCHES ET MERGES AVANCÉS

Trouver de l'aide pour résoudre les conflits durant un merge

Tous les changements que git peut merger automatiquement sont déjà ajoutés à l'index, donc git diff ne vous montre que les conflits. Il a une syntaxe peu commune:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD:file.txt
 +Hello world
++=======
+ Goodbye
++>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

Souvenez-vous que le commit qui sera effectué après que nous ayons résolu ce conflit aura 2 parents: l'un sera le sommet de la branche courante (HEAD) et l'autre sera le sommet de la branche qui s'occupe du merge, stockée temporairement dans MERGE_HEAD.

Durant le merge, l'index garde 3 versions de chaque fichiers. Une de ces 3 "étape de fichier" représente une version différente du fichier:

```
$ git show :1:fichier.txt   # Le fichier dans l'ancêtre commun des 2 branches
$ git show :2:fichier.txt   # La version présente dans HEAD.
$ git show :3:fichier.txt   # La version présente dans MERGE_HEAD.
```

Quand vous demandez à git diff de vous montrer les conflits, il fait une différence en 3 points entre les résultats conflictuels de merge dans le répertoire de travail avec les version 2 et 3 pour montrer seulement les morceaux de code qui ont du contenu de chaque côté, mélangés (en d'autres termes, quand un morceau du résultat du merge ne vient que de la version 2, alors ce morceau n'est pas en conflit est n'est pas affiché. Idem pour la version 3).

La différence en début de chapitre vous montre les différences entre la version de travail de fichier.txt et les versions 2 et 3. Donc au lieux de rajouter les préfixes "+" ou "-" devant chaque ligne, on utilise maintenant 2 colonnes pour ces préfixes: la première colonne est utilisée pour les différences entre le premier parent et la copie du répertoire de travail, et la deuxième pour les différences entre le second parent et la copie du répertoire de travail. (Voir la section "COMBINED DIFF FORMAT" dans la documentation de git diff-files pour plus de détails sur ce format.)

Après avoir résolu le conflit de manière évidente (mais avant de mettre à jour l'index), le diff ressemblera à:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,1 @@@
- Hello world
-Goodbye
++Goodbye world
```

Cela montre que notre version résolue a effacé "Hello world" du premier parent, et effacé "Goodbye" du second parent, puis ajouté "Goodbye world", qui était avant absent des 2.

Quelques options spéciales de diff permettent de faire la différence entre le répertoire de travail et les différentes étapes:

```
$ git diff -1 file.txt          # différence avec l'étape 1
$ git diff --base file.txt      # même chose que ci-dessus
$ git diff -2 file.txt          # différence avec l'étape 2
$ git diff --ours file.txt      # même chose que ci-dessus
$ git diff -3 file.txt          # différence avec l'étape 3
$ git diff --theirs file.txt    # même chose que ci-dessus
```

Les commandes git log and gitk fournissent aussi de l'aide particulière pour les merges:

```
$ git log --merge
$ gitk --merge
```

Cela vous montrera tous les commits qui existent seulement des HEAD ou dans MERGE_HEAD, et qui concernant les fichiers non-mergés.

Vous pouvez aussi utiliser git mergetool, qui vous permet de merger des fichiers non-mergés en utilisant des outils externes comme emacs ou kdiff3.

Chaque fois que vous résolvez les conflits d'un fichier et que vous mettez à jour l'index:

```
$ git add fichier.txt
```

les différentes étapes de se fichiers "s'effondreront", après quoi git-diff ne montrera plus (par défaut) de différence pour ce fichier.

Merge Multiples

Vous pouvez combiner plusieurs branches en même temps en les listant simplement dans la même commande git merge. Par exemple:

```
$ git merge scott/master rick/master tom/master
```

est l'équivalent de:

```
$ git merge scott/master
$ git merge rick/master
$ git merge tom/master
```

Subtree

Il y a des moments où vous voulez inclure le contenu d'un projet développé indépendamment dans votre projet. Vous pouvez juste récupérer le code de cet autre projet tant que les chemins ne rentrent pas en conflit.

Les problèmes surviennent quand il y a des fichiers en conflit. Les candidats potentiels sont les Makefiles et les autres noms de fichiers standardisés. Vous pouvez merger ces fichiers, mais ce n'est pas forcément ce que vous voulez faire. Une meilleure solution pour ce problème peut être de merger le projet dans un sous-répertoire. Mais la stratégie de merge ne supporte la récursivité, donc la simple récupération des fichiers ne fonctionnera pas.

Vous voudrez alors utiliser la stratégie de merge subtree, qui vous aidera dans cette situation.

Dans cet exemple, disons que vous avez un dépôt dans /path/to/B (si vous voulez ça peut aussi être une adresse URL). Vous voulez merger la branche "master" de ce dépôt dans le sous-répertoire dir-B de la branche actuelle.

Voici la séquence de commande que vous utiliserez:

```
$ git remote add -f Bproject /path/to/B (1)
$ git merge -s ours --no-commit Bproject/master (2)
$ git read-tree --prefix=dir-B/ -u Bproject/master (3)
$ git commit -m "Merge B project as our subdirectory" (4)
$ git pull -s subtree Bproject master (5)
```

Utiliser le merge subtree vous permet d'apporter moins de complications administratives aux les utilisateurs de votre dépôt. Elle est aussi compatible avec des version plus anciennes de git (jusqu'à Git v1.5.2) et vous aurez le code juste après le clonage.

Cependant, si vous utilisez des sous-modules, vous pouvez alors choisir de ne pas transférer les objets de ces sous-modules. Cela peut poser un problème lors de l'utilisation des merge subtree.

Il est aussi plus facile d'effectuer des changement dans votre autre projet si vous utiliser les sous-modules.

(from Using Subtree Merge)

TROUVER LES PROBLÈMES - GIT BISECT

Supposons que la version 2.6.18 de votre project fonctionne correctement, mais que la version sur la branche "master" plante. Parfois, la meilleure façon de trouver la cause d'une telle régression est de faire de la recherche brutale dans l'historique de votre projet pour trouver un commit en particulier responsable du problème. La commande git bisect peut vous aider dans cette démarche:

```
$ git bisect start
$ git bisect good v2.6.18
$ git bisect bad master
Bisecting: 3537 revisions left to test after this
[65934a9a028b88e83e2b0f8b36618fe503349f8e] BLOCK: Make USB storage depend on SCSI rather than selecting it [try
```

Si vous lancez "git branch" à ce moment, vous verrez que git vous a temporairement déplacé sur une nouvelle branche nommée "bisect". Cette branche pointe vers un commit (avec l'identifiant 65934...) qui est accessible depuis "master" mais pas depuis v2.6.18. Compilez et testez votre projet pour voir quand il plante. En assument qu'il plante. Ensuite:

```
$ git bisect bad
Bisecting: 1769 revisions left to test after this
[7eff82c8b1511017ae605f0c99ac275a7e21b867] i2c-core: Drop useless bitmaskings
```

récupère une version plus ancienne. Continuez comme ça, en disant à chaque fois à git sit la version qu'il vous donne est bonne ou mauvaise, et divisant par deux le nombre de révisions à tester à chaque étape.

Après environ 13 test (dans ce cas), il vous montrera l'identifiant du commit qui pose problème. Vous pouvez examiner ce commit avec git show, trouver qui a publié ce commit, et lui envoyer votre rapport de bug par mail avec l'identifiant du commit concerné. Pour terminer, lancez:

```
$ git bisect reset
```

pour revenir à la branche sur laquelle vous vous trouviez et effacer la branche temporaire "bisect".

La version que git-bisect vous récupère à chaque étape n'est qu'une suggestion, et vous pouvez choisir une version différente si vous pensez que c'est une meilleure idée. Par exemple, vous tomberez parfois sur un commit qui pose un problème qui ne concerne pas notre débuggage; lancez:

```
$ git bisect visualize
```

qui affichera gitk et marquera les commits qu'il choisira avec une étiquette "bisect". Choisissez un commit qui vous parait sûr, notez l'identifiant du commit, et récupérez le avec:

```
$ git reset --hard fb47ddb2db...
```

puis testez, lancez "bisect good" ou "bisect bad" comme il convient, et continuez.

TROUVER LES PROBLÈMES - GIT BLAME

La commande linkto:git-blame[1] est vraiment utile pour savoir qui a modifié quelle partie du fichier. Si vus lancez simplement 'git blame [nom_du_fichier]' vous obtiendrez une liste du dernier sha du commit, de la date et de l'auteur de chaque ligne du fichier.

```
$ git blame sha1_file.c
...
0fcfd160 (Linus Torvalds  2005-04-18 13:04:43 -0700    8)  */
0fcfd160 (Linus Torvalds  2005-04-18 13:04:43 -0700    9) #include "cache.h"
1f688557 (Junio C Hamano  2005-06-27 03:35:33 -0700   10) #include "delta.h"
a733cb60 (Linus Torvalds  2005-06-28 14:21:02 -0700   11) #include "pack.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   12) #include "blob.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   13) #include "commit.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   14) #include "tag.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   15) #include "tree.h"
f35a6d3b (Linus Torvalds  2007-04-09 21:20:29 -0700   16) #include "refs.h"
70f5d5d3 (Nicolas Pitre   2008-02-28 00:25:19 -0500   17) #include "pack-revindex.h"628522ec (Junio C Hamano
...
```

C'est souvent utile pour voir qui a crée une erreur si le fichier revient sur une ligne ou qu'une erreur casse la compilation.

Vous pouvez aussi sélectionner le numéro des ligne de début et de fin de la partie de fichier à analyser:

```
$>git blame -L 160,+10 sha1_file.c
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       160)}
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       161)
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700       162)/*
```

```
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700       163) * NOTE! This returns a statically allocate
790296fd (Jim Meyering   2008-01-03 15:18:07 +0100       164) * careful about using it. Do an "xstrdup()
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700       165) * filename.
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       166) *
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       167) * Also note that this returns the location
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       168) * SHA1 file can happen from any alternate
d19938ab (Junio C Hamano 2005-05-09 17:57:56 -0700       169) * DB_ENVIRONMENT environment variable if i
```

GIT ET LES MAILS

Envoyer des patches pour un projet

Si vous avez juste effectué quelques modifications, le mail est la manière la plus simple d'envoyer ces changements comme patches:

D'abord, utilisez git format-patch; par exemple:

```
$ git format-patch origin
```

produira un liste numérotée de fichiers présent dans le répertoire actuel, avec un élément de la liste pour chaque patch de la branche courante qui n'est pas présent dans la branche d'origine distante (origin/HEAD).

Vous pouvez ensuite importer ce texte dans votre client mail, et l'envoyer manuellement. Cependant, si vous avez beaucoup de patches à envoyer en même temps, il est préférable d'utiliser le script git send-email pour automatiser le processus. Consultez la mailing-list du projet concerné pour savoir comment ces patches sont habituellement gérés.

Importer les patches dans un project

Git fournit aussi un outil nommée git am; (am signifiant "apply mailbox"), pour important un série de mails contenant des patches. Sauvegardez tous les messages des mails contenant les patches, dans l'ordre, dans une seule boite mail, disons "patches-mbox", et lancez:

```
$ git am -3 patches.mbox
```

Git appliquera chaque patch dans l'ordre; quand un conflit est trouvé, il s'arrêtera et vous pourrez réparer les conflits comme décrit dans "<<resolving-a-merge,Résoudre un merge>>". (L'option "-3" dit à git d'effectuer un merge; si vous préférer juste laisser votre branche et index inchangés, vous pouvez omettre cette option).

Une fois que l'index est mis à jour avec la résolution du conflit, au lieu de créer un nouveau commit, lancez juste:

```
$ git am --resolved
```

et git créera un commit pour vous, et continuera d'appliquer les patches restant dans la boite mail.

Le résultat final sera une série de commits, un par patch présent dans la boite mail originale, avec l'auteur et le message du commit venant du mail correspondant à chaque patch.

PERSONNALISER GIT

git config

Changer l'Editeur

```
$ git config --global core.editor emacs
```

Ajouter des Alias

```
$ git config --global alias.last 'cat-file commit HEAD'

$ git last
tree c85fbd1996b8e7e5eda1288b56042c0cdb91836b
parent cdc9a0a28173b6ba4aca00eb34f5aabb39980735
author Scott Chacon <schacon@gmail.com> 1220473867 -0700
committer Scott Chacon <schacon@gmail.com> 1220473867 -0700

fixed a weird formatting problem

$ git cat-file commit HEAD
tree c85fbd1996b8e7e5eda1288b56042c0cdb91836b
parent cdc9a0a28173b6ba4aca00eb34f5aabb39980735
author Scott Chacon <schacon@gmail.com> 1220473867 -0700
committer Scott Chacon <schacon@gmail.com> 1220473867 -0700

fixed a weird formatting problem
```

Ajouter de la Couleur

Voir toutes les options color.* dans la documentation de git config

```
$ git config color.branch auto
$ git config color.diff auto
$ git config color.interactive auto
$ git config color.status auto
```

Ou, vous pouvez tous les activer avec l'option color.ui:

```
$ git config color.ui true
```

Template de Commit

```
$ git config commit.template '/etc/git-commit-template'
```

Format de Log

```
$ git config format.pretty oneline
```

Autres Options de Configuration

Il y a aussi un grand nombre d'options intéressantes pour emballer, nettoyer, merger, créer des branches, pour le transport http, les diff, et plus. Si vous voulez personnaliser toutes ces options, reportez-vous à la documentation de git config.

GIT HOOKS

Git Hooks

Hooks Côté Serveur

Reception de Post

```
GIT_DIR/hooks/post-receive
```

Si vous écrivez ceci en Ruby, vous pourrez obtenir les arguments de cette manière:

```
rev_old, rev_new, ref = STDIN.read.split(" ")
```

Ou en script bash, quelque chose comme ça peut fonctionner:

```
#!/bin/sh
# <oldrev> <newrev> <refname>
# update a blame tree
while read oldrev newrev ref
do
    echo "STARTING [$oldrev $newrev $ref]"
    for path in `git diff-tree -r $oldrev..$newrev | awk '{print $6}'`
    do
      echo "git update-ref refs/blametree/$ref/$path $newrev"
      `git update-ref refs/blametree/$ref/$path $newrev`
    done
done
```

Hooks Côté Client

Pre Commit

Lancer vos tests automatiquement avant de committer

```
GIT_DIR/hooks/pre-commit
```

Voici un exemple d'un script Ruby qui exécute des tests RSpec avant de vous permettre de faire un commit.

```
html_path = "spec_results.html"
`spec -f h:#{html_path} -f p spec` # lancer rspec, affichier la prograssion, sauvegarde le resul
```

```
# affiche combien d'erreurs ont été trouvées
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
failures = html.match(/(\d+) failures/)[0].to_i rescue 0
pending = html.match(/(\d+) pending/)[0].to_i rescue 0

if failures.zero?
  puts "0 failures! #{examples} run, #{pending} pending"
else
  puts "\aDID NOT COMMIT YOUR FILES!"
  puts "View spec results at #{File.expand_path(html_path)}"
  puts
  puts "#{failures} failures! #{examples} run, #{pending} pending"
  exit 1
end
```

- http://probablycorey.wordpress.com/2008/03/07/git-hooks-make-me-giddy/

RETROUVER DES OBJETS CORROMPUS

Retrouver les derniers commits - Article de Blog (en)

Retrouver des Blob Corrompus par Linus

SUBMODULES

Large projects are often composed of smaller, self-contained modules. For example, an embedded Linux distribution's source tree would include every piece of software in the distribution with some local modifications; a movie player might

need to build against a specific, known-working version of a decompression library; several independent programs might all share the same build scripts.

With centralized revision control systems this is often accomplished by including every module in one single repository. Developers can check out all modules or only the modules they need to work with. They can even modify files across several modules in a single commit while moving things around or updating APIs and translations.

Git does not allow partial checkouts, so duplicating this approach in Git would force developers to keep a local copy of modules they are not interested in touching. Commits in an enormous checkout would be slower than you'd expect as Git would have to scan every directory for changes. If modules have a lot of local history, clones would take forever.

On the plus side, distributed revision control systems can much better integrate with external sources. In a centralized model, a single arbitrary snapshot of the external project is exported from its own revision control and then imported into the local revision control on a vendor branch. All the history is hidden. With distributed revision control you can clone the entire external history and much more easily follow development and re-merge local changes.

Git's submodule support allows a repository to contain, as a subdirectory, a checkout of an external project. Submodules maintain their own identity; the submodule support just stores the submodule repository location and commit ID, so other developers who clone the containing project ("superproject") can easily clone all the submodules at the same revision. Partial checkouts of the superproject are possible: you can tell Git to clone none, some or all of the submodules.

The git submodule command is available since Git 1.5.3. Users with Git 1.5.2 can look up the submodule commits in the repository and manually check them out; earlier versions won't recognize the submodules at all.

To see how submodule support works, create (for example) four example repositories that can be used later as a submodule:

```
$ mkdir ~/git
$ cd ~/git
```

```
$ for i in a b c d
do
    mkdir $i
    cd $i
    git init
    echo "module $i" > $i.txt
    git add $i.txt
    git commit -m "Initial commit, submodule $i"
    cd ..
done
```

Now create the superproject and add all the submodules:

```
$ mkdir super
$ cd super
$ git init
$ for i in a b c d
do
    git submodule add ~/git/$i
done
```

NOTE: Do not use local URLs here if you plan to publish your superproject!

See what files `git-submodule` created:

```
$ ls -a
.  ..  .git  .gitmodules  a  b  c  d
```

The `git-submodule add` command does a couple of things:

- It clones the submodule under the current directory and by default checks out the master branch.
- It adds the submodule's clone path to the gitmodules file and adds this file to the index, ready to be committed.

- It adds the submodule's current commit ID to the index, ready to be committed.

Commit the superproject:

```
$ git commit -m "Add submodules a, b, c and d."
```

Now clone the superproject:

```
$ cd ..
$ git clone super cloned
$ cd cloned
```

The submodule directories are there, but they're empty:

```
$ ls -a a
.  ..
$ git submodule status
-d266b9873ad50488163457f025db7cdd9683d88b a
-e81d457da15309b4fef4249aba9b50187999670d b
-c1536a972b9affea0f16e0680ba87332dc059146 c
-d96249ff5d57de5de093e6baff9e0aafa5276a74 d
```

NOTE: The commit object names shown above would be different for you, but they should match the HEAD commit object names of your repositories. You can check it by running `git ls-remote ../a`.

Pulling down the submodules is a two-step process. First run `git submodule init` to add the submodule repository URLs to `.git/config`:

```
$ git submodule init
```

Now use `git-submodule update` to clone the repositories and check out the commits specified in the superproject:

```
$ git submodule update
$ cd a
$ ls -a
.  ..  .git  a.txt
```

One major difference between `git-submodule update` and `git-submodule add` is that `git-submodule update` checks out a specific commit, rather than the tip of a branch. It's like checking out a tag: the head is detached, so you're not working on a branch.

```
$ git branch
* (no branch)
master
```

If you want to make a change within a submodule and you have a detached head, then you should create or checkout a branch, make your changes, publish the change within the submodule, and then update the superproject to reference the new commit:

```
$ git checkout master
```

or

```
$ git checkout -b fix-up
```

then

```
$ echo "adding a line again" >> a.txt
$ git commit -a -m "Updated the submodule from within the superproject."
$ git push
$ cd ..
$ git diff
diff --git a/a b/a
```

```
index d266b98..261dfac 160000
--- a/a
+++ b/a
@@ -1 +1 @@
-Subproject commit d266b9873ad50488163457f025db7cdd9683d88b
+Subproject commit 261dfac35cb99d380eb966e102c1197139f7fa24
$ git add a
$ git commit -m "Updated submodule a."
$ git push
```

You have to run `git submodule update` after `git pull` if you want to update submodules, too.

Pitfalls with submodules

Always publish the submodule change before publishing the change to the superproject that references it. If you forget to publish the submodule change, others won't be able to clone the repository:

```
$ cd ~/git/super/a
$ echo i added another line to this file >> a.txt
$ git commit -a -m "doing it wrong this time"
$ cd ..
$ git add a
$ git commit -m "Updated submodule a again."
$ git push
$ cd ~/git/cloned
$ git pull
$ git submodule update
error: pathspec '261dfac35cb99d380eb966e102c1197139f7fa24' did not match any file(s) known to git.
Did you forget to 'git add'?
Unable to checkout '261dfac35cb99d380eb966e102c1197139f7fa24' in submodule path 'a'
```

You also should not rewind branches in a submodule beyond commits that were ever recorded in any superproject.

It's not safe to run `git submodule update` if you've made and committed changes within a submodule without checking out a branch first. They will be silently overwritten:

```
$ cat a.txt
module a
$ echo line added from private2 >> a.txt
$ git commit -a -m "line added inside private2"
$ cd ..
$ git submodule update
Submodule path 'a': checked out 'd266b9873ad50488163457f025db7cdd9683d88b'
$ cd a
$ cat a.txt
module a
```

NOTE: The changes are still visible in the submodule's reflog.

This is not the case if you did not commit your changes.

Chapter 6

Travailler avec Git

GIT SUR WINDOWS

(mSysGit)

DÉPLOYER AVEC GIT

Capistrano et Git

GitHub Guide on Deploying with Cap

Git and Capistrano Screencast

INTÉGRATION DE SUBVERSION

MIGRATION DEPUIS UN AUTRE SCM

Vous avez décidé de migrez depuis un autre système de contrôle de version et convertir l'intégralité de votre projet vers Git. Comment le faire facilement?

Importation depuis Subversion

Git est fournit avec un script nommé git-svn qui est une commande de clonage qui importe un dépôt subversion dans un nouveau dépôt git. Il existe aussi un outil gratuit sur le service GitHub qui fera cette manipulation pour vous.

```
$ git-svn clone http://my-project.googlecode.com/svn/trunk new-project
```

Ceci vous donnera un nouveau dépôt Git avec tout l'historique du dépôt Subversion original. Cela peut prendre beaucoup de temps, puisque qu'il démarre avec la version 1 du projet subversion puis vérifie et commit localement chaque révision une par une.

Importation depuis Perforce

Dans le répertoire contrib/fast-import, vous trouverez le script git-p4. C'est un script python qui importera un dépôt Perforce pour vous.

```
$ ~/git.git/contrib/fast-import/git-p4 clone //depot/project/main@all myproject
```

Importation depuis un autre SCM

Il y a d'autres SCMs listés en tête du Sondage Git, de la documentation est nécesaire pour eux. !!TODO!!

- CVS
- Mercurial (hg)
- Bazaar-NG
- Darcs
- ClearCase

GIT EN MODE GRAPHIQUE

Git possède quelques interfaces graphiques assez populaires. Elles permettent de lire et/ou manipuler des dépôt Git.

Les Interfaces Fournies

Git est fourni avec 2 programmes graphiques écrits en Tcl/Tk. Gitk est un navigateur de dépôt et un visualiseur d'historique de commits.

gitk

git gui est un outil qui vous aidera à visualiser les opérations sur l'index, comme les ajouts, les retraits et les commits. Il ne permet pas faire toutes les opérations disponibles avec la ligne de commande, mais pour les opérations les plus basiques, c'est un très bon outil.

git gui

Projets Tiers

Pour les utilisateurs de Mac, vous trouverez GitX et GitNub

Pour Linux et les autres utilisateurs de Qt, il y a QGit

HÉBERGEMENT GIT

github

repoorcz

USAGES ALTERNATIFS

ContentDistribution

TicGit

LE LANGAGE DE SCRIPT ET GIT

Ruby et Git

grit

jgit + jruby

PHP et Git

Python et Git

pygit

Perl et Git

perlgit

GIT ET LES EDITEURS DE CODE

textmate

eclipse

netbeans

Chapter 7

Fonctionnement Interne et Plomberie

COMMENT GIT STOCKE LES OBJETS

Ce chapitre s'enfouit dans les détails concernant le stockage physique des objets avec Git.

Tous les objets sont stockés comme du contenu comprimés par leur valeur SHA. Il contiennent le type d'objet, la taille et le contenu au format gzip.

Git garde les objets dans 2 formats: les objets détendus et les objets packagés.

Les Objets Détendus

Le format le plus simple concerne les objets détendus. Ce n'est que des données comprimées stockées dans un seul fichier sur le disque. Chaque objet est écrit dans un fichier séparé.

Si le SHA de l'objet est ab04d884140f7b0cf8bbf86d6883869f16a46f65, alors le fichier sera stocké dans le chemin suivant:

```
GIT_DIR/objects/ab/04d884140f7b0cf8bbf86d6883869f16a46f65
```

Il récupère les 2 premiers caractères et les utilisent comme sous-répertoire, comme ça il n'y a jamais trop d'objets dans le même dossier. Le nom du fichier est alors composé des 38 caractères restant.

En utilisant cette implémentation Ruby du stockage avec Git, nous pouvons examiner simplement comment les données d'un objet sont stockées:

```
def put_raw_object(content, type)
  size = content.length.to_s

  header = "#{type} #{size}#body"
  store = header + content

  sha1 = Digest::SHA1.hexdigest(store)
  path = @git_dir + '/' + sha1[0...2] + '/' + sha1[2..40]

  if !File.exists?(path)
    content = Zlib::Deflate.deflate(store)

    FileUtils.mkdir_p(@directory+'/'+sha1[0...2])
    File.open(path, 'w') do |f|
      f.write content
    end
  end
  return sha1
end
```

Les Objets Packagés

Le packfile est l'autre format de stockage d'objet. Depuis que Git stocke chaque version de chaque fichier dans un objet séparé, il peu rapidement devenir inefficace avec le format précédent Imaginez avoir un fichier long de quelques milliers de lignes et si on change une ligne, git enregistrera un second fichier dans son intégrité, cela serait un grand gaspillage d'espace.

Afin de sauver cet espace, Git utilise le packfile. C'est une format où Git ne va sauvegarder que la partie du fichier qui a changé dans le second fichier, avec un pointeur vers le fichier similaire.

Quand les objets sont écrits sur le disque, le format détendu est souvent utilisé, puisque ce format est plus simple d'accès. Cependant, vous voudrez certainement sauver de l'espace en packageant les objets - cela se fait avec la commande git gc. Elle utilisera une heuristique plutôt compliquée pour déterminer quels fichiers sont similaires et basera les delta sur cette analyse. Nous pouvons avoir de multiple packfiles, ils peuvent être re-packagés si nécessaire (git repack) ou dé-packagés en de multiples fichiers (git unpack-objects) relativement facilement.

Git écrira aussi un fichier d'index pour chaque packfile, il sera bien plus petit et contiendra les offset du packfile pour trouver des objets spécifiques plus rapidement en utilisant le SHA.

Le détail de l'implémentation du packfile se trouve dans le chapitre Packfile disponible un peu plus loin dans ce livre.

NAVIGATION DANS LES OBJETS GIT

Nous pouvons demander à git des informations à propos d'un objet particulier avec la commande cat-file. Vous pouvez utilise le SHA partiel pour éviter de saisir les 40 caractères:

```
$ git-cat-file -t 54196cc2
commit
$ git-cat-file commit 54196cc2
tree 92b8b694ffb1675e5975148e1121810081dbdffe
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500

initial commit
```

Un "tree" peut référencer un ou plusieurs objets "blob", chacun correspondant à un fichier. En plus, un "tree" peut aussi référencer d'autres objets "tree", créant ainsi une hiérarchie de dossier. Vous pouvez examiner le contenu de n'importe quel "tree" en utilisant ls-tree (souvenez-vous qu'une portion suffisamment longue du SHA1 fonctionnera aussi):

```
$ git ls-tree 92b8b694
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    file.txt
```

Nous pouvons voir que ce tree contient un fichier. Le hash SHA1 est une référence aux données de ce fichier:

```
$ git cat-file -t 3b18e512
blob
```

Un "blob" n'est qu'un fichier de donnée, qui peut être examiné avec cat-file:

```
$ git cat-file blob 3b18e512
hello world
```

Notez qu'il s'agit ici des données de l'ancien fichier; donc l'objet que git appelle dans sa réponse au "tree" initial était un "tree" avec une capture de l'état du dossier qui a été enregistré par le premier commit.

Tous les objets sont stockés sous leur nom SHA1 dans le répertoire git:

```
$ find .git/objects/
.git/objects/
.git/objects/pack
.git/objects/info
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/92
.git/objects/92/b8b694ffb1675e5975148e1121810081dbdffe
.git/objects/54
.git/objects/54/196cc2703dc165cbd373a65a4dcf22d50ae7f7
.git/objects/a0
.git/objects/a0/423896973644771497bdc03eb99d5281615b51
.git/objects/d0
.git/objects/d0/492b368b66bdabf2ac1fd8c92b39d3db916e59
.git/objects/c4
.git/objects/c4/d59f390b9cfd4318117afde11d601c1085f241
```

et le contenu de ces fichiers n'est que des données comprimées avec une entête identifiant leur type et leur taille. Le type peut être un blob, un tree, un commit, ou un tag.

Le commit le plus simple à trouver est le commit HEAD, qui peut être trouvé dans .git/HEAD:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Comme vous pouvez le voir, ceci nous dit dans quelle branche nous nous trouvons, et il nous le dit en nommant le fichier dans le répertoire .git, qui lui-même contient le nom SHA1 référant à l'objet commit, qui peut être examiné avec cat-file:

```
$ cat .git/refs/heads/master
c4d59f390b9cfd4318117afde11d601c1085f241
$ git cat-file -t c4d59f39
commit
$ git cat-file commit c4d59f39
```

```
tree d0492b368b66bdabf2ac1fd8c92b39d3db916e59
parent 54196cc2703dc165cbd373a65a4dcf22d50ae7f7
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143418702 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143418702 -0500

add emphasis
```

L'objet "tree" réfère ici au nouvel état du "tree":

```
$ git ls-tree d0492b36
100644 blob a0423896973644771497bdc03eb99d5281615b51    file.txt
$ git cat-file blob a0423896
hello world!
```

et l'objet "parent" réfère au commit précédent:

```
$ git-cat-file commit 54196cc2
tree 92b8b694ffb1675e5975148e1121810081dbdffe
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
```

LES RÉFÉRENCES GIT

Les branches, les branches de suivie distant, et les tags sont tous des références à des commits. Toutes les références sont nommées avec un nom de chemin séparé par des slash commençant par "refs"; Les noms que nous avons utilisés jusqu'à maintenant sont actuellement des raccourcis:

```
- La branche "test" est un raccourci pour "refs/heads/test".
- Le tag "v2.6.18" est un raccourci pour "refs/tags/v2.6.18".
- "origin/master" est un raccourci pour "refs/remotes/origin/master".
```

Le nom complet est parfois utile si, par exemple, il existe un tag et une branche avec le même nom.

(les références fraîchement créées sont stockées dans le dossier .git/refs, sous le chemin donné par leur nom. Cependant, pour plus d'efficacité, elles peuvent aussi être packagées ensemble dans un seul fichier; voir git pack-refs).

Un autre raccourci utile, le "HEAD" d'un dépôt peut être référencé en utilisant simplement le nom de ce dépôt. PAr exemple, "origin" est généralement utilisé comme raccourci vers la branche HEAD dans le dot "origin".

Pour la liste complète des chemins que git utilise comme référence, et l'ordre utilisé pour décider lequel choisir quand il y a plusieurs référence avec le même raccourci, voir la section "SPECIFYING REVISIONS" de git rev-parse.

Montrer les commits présents uniquement dans une branche

Supposons que vous voulez voir tous les commits atteignable depuis la branche centrale nommée "master" mais non-visible depuis les autres branches de votre dépôt.

Nous pouvons faire la liste des sommets du dépôts avec git show-ref:

```
$ git show-ref --heads
bf62196b5e363d73353a9dcf094c59595f3153b7 refs/heads/core-tutorial
db768d5504c1bb46f63ee9d6e1772bd047e05bf9 refs/heads/maint
a07157ac624b2524a059a3414e99f6f44bebc1e7 refs/heads/master
24dbc180ea14dc1aebe09f14c8ecf32010690627 refs/heads/tutorial-2
1e87486ae06626c2f31eaa63d26fc0fd646c8af2 refs/heads/tutorial-fixes
```

Nous pouvons afficher seulement le nom de la branche, et enlever "master", avec l'aide des outils standards comme cut et grep:

```
$ git show-ref --heads | cut -d' ' -f2 | grep -v '^refs/heads/master'
refs/heads/core-tutorial
refs/heads/maint
refs/heads/tutorial-2
refs/heads/tutorial-fixes
```

Et ensuite nous pouvons demander à voir tous les commits visibles depuis "master" mais pas des autres sommets:

```
$ gitk master --not $( git show-ref --heads | cut -d' ' -f2 |
                grep -v '^refs/heads/master' )
```

Évidemment, des variations illimités sont possibles, par exemple, pour afficher tous les commits visible de certains sommets mais pas de ceux contenant des tags dans le dépôt:

```
$ gitk $( git show-ref --heads ) --not  $( git show-ref --tags )
```

(Voir git rev-parse pour plus d'explications sur la syntaxe de sélection de commits comme `--not`.)

(!!update-ref!!)

L'INDEX GIT

L'index est un fichier binaire (généralement gardé dans .git/index) qui contient une liste ordonnées de chemins de fichiers, chacun avec les permissions et le SHA1 de l'objet blob; git ls-files peut vous montrer le contenu de l'index:

```
$ git ls-files --stage
100644 63c918c667fa005ff12ad89437f2fdc80926e21c 0   .gitignore
100644 5529b198e8d14decbe4ad99db3f7fb632de0439d 0   .mailmap
100644 6ff87c4664981e4397625791c8ea3bbb5f2279a3 0   COPYING
100644 a37b2152bd26be2c2289e1f57a292534a51a93c7 0   Documentation/.gitignore
```

```
100644 fbefe9a45b00a54b58d94d06eca48b03d40a50e0 0   Documentation/Makefile
...
100644 2511aef8d89ab52be5ec6a5e46236b4b6bcd07ea 0   xdiff/xtypes.h
100644 2ade97b2574a9f77e7ae4002a4e07a6a38e46d07 0   xdiff/xutils.c
100644 d5de8292e05e7c36c4b68857c1cf9855e3d2f70a 0   xdiff/xutils.h
```

Dans une documentation plus ancienne, vous pourrez voir l'index appelé "cache du répertoire" ou juste "cache". L'index comporte 3 propriétés importantes:

1. L'index contient toutes les informations nécessaire pour générer un unique (déterminé uniquement) objet "tree".

   Par exemple, le lancement de git commit génère un objet tree depuis l'index, le stocke dans la base de donnée objet, et l'utilise comme l'objet tree associé au nouveau commit.

2. L'index permet une comparaison rapide entre les objet tree qu'il définit et le tree de travail.

   Il fait ceci en stockant des informations complémentaires pour chaque entrée (comme la date de dernière modification). Ces données ne sont pas affichées ci-dessus, et ne sont pas stockées dans l'objet tree créé, mais elles peuvent être utilisées pour déterminer rapidement quels fichiers du répertoire du travail diffèrent de ce que sont stockés dans l'index, et donc permet à git de gagner du temps sans avoir besoin de lire toutes les données pour ce genre de fichiers pour connaître les changements

3. Il peut être efficacement représenter les information à propos des conflits de merge entre différentes version des objets tree, permettant à chaque chemin de fichier d´être associé avec suffisament d'information à propos des "tree" impliqués dans la création d'un merge three-way.

   Nous avons vu dans <<conflict-resolution>> que durant le merge, l'index peut stocker de multiples version d'un même fichier (appelés "stages"). La troisième colonne dans la sortie de git ls-files ci-dessus est le numéro du "stage", et prendra une valeur autre que 0 pour les fichiers avec des conflits de merge.

L'index est donc un sorte de zone d'assemblage temporaire, qui contient le "tree" sur lequel vous êtes en train de travailler.

LE PACKFILE

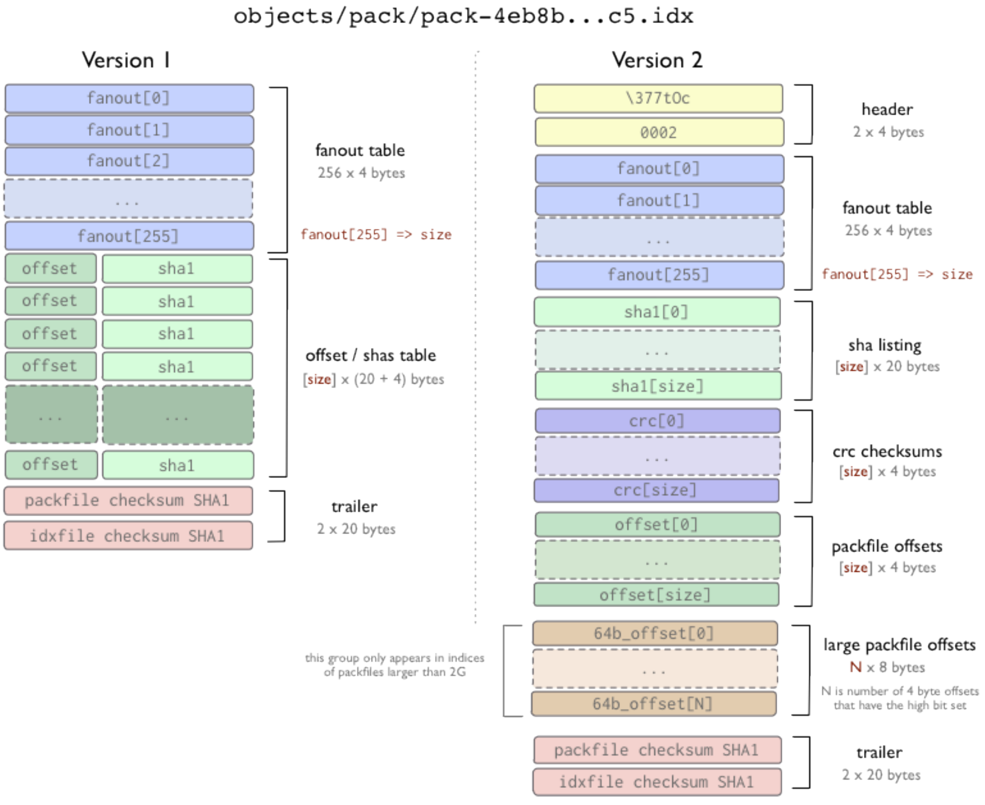
Ce chapitre explique en détails, au bit près, comment le packfile et les fichier d'index de pack sont formatés.

L'Index Packfile

Premièrement, nous avons l'index packfile, qui est simplement juste une série de marque-pages vers le packfile.

Il existe 2 versions de l'index packfile - version 1, qui est celle par défaut dans les version de Git antérieures à 1.6, et la version 2, qui est celle par défaut depuis Git v1.6, et qui peut être lue par Git jusqu'à la v1.5.2, et qui a aussi était portée sur la v1.4.4.5 sur vous travaillez toujours dans la série 1.4.

La version 2 inclue un checksum CRC pour chaque objet afin que les données compressées puissent être copiées directement entre les pack durant les re-package sans se retrouver avec de données corrumpues non-détectées. Les index de version 2 peuvent aussi s'occuper de packfiles supérieurs à 4Go.



Dans les 2 formats, la table de routage (fanout) est simplement un manière plus rapide de trouver dans le fichier d'index l'offset d'un SHA particulier. Les tables offset/sha1[] sont triées par valeurs sha1[] (cela permet les recherches binaires sur cette table), et la table fanout[] pointe vers la table offset/sha1[] de façon spécifique (pour que la part de la dernière table

qui convertie tous les hash commençant par un bit en particulier afin qu'ils puissent être trouvés en évitant les 8 itérations de la recherche binaire).

Dans la version 1, les offsets et SHAs sont au même endroit, alors que dans la version 2, il y a des tables séparés pour les SHAs, les checksums CRC et les offsets. Les checksums SHAs pour le fichier d'index et le packfile se trouvent à la fin de chacun de ces fichiers.

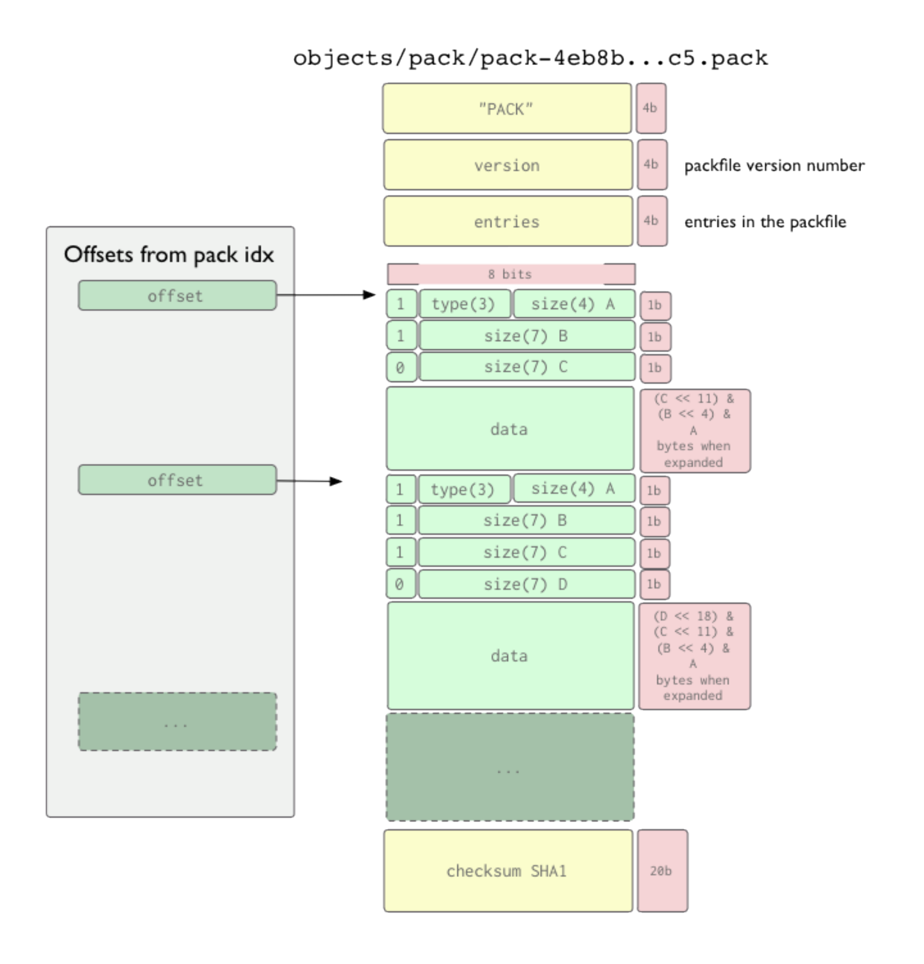
Il es important de noter que les index de packfile *ne sont pas* nécessaires pour extraire les objets d'un packfile, ils sont juste utilisé pour retrouver *rapidement* des objets individuels depuis un pack. Le format du packfile est utilisé dans les programmes upload-pack et receive-pack (protocoles de publication - push - et de récupération - fetch) pour transférer les objets, et il n'y a alors pas d'index utilisé - il peut être construit après coup en scannant le packfile.

Le Format du Packfile

Le format du packfile même est très simple. Il y a une entête, une série d'objets packagés (chacun avec sa propre entête et son propre corps), puis un checksum à la fin. Les 4 premiers bits forment a chaîne de caractère "PACK", qui est utilisée pour être sûr que vous lisez correctement le début du packfile. Ceci est suivi d'un numéro de version du packfile sur 4 bits, puis 4 autres bis représentant le nombre d'entrées dans ce fichier. En Ruby, vous pourriez lire les données de l'entête de cette manière:

```
def read_pack_header
  sig = @session.recv(4)
  ver = @session.recv(4).unpack("N")[0]
  entries = @session.recv(4).unpack("N")[0]
  [sig, ver, entries]
end
```

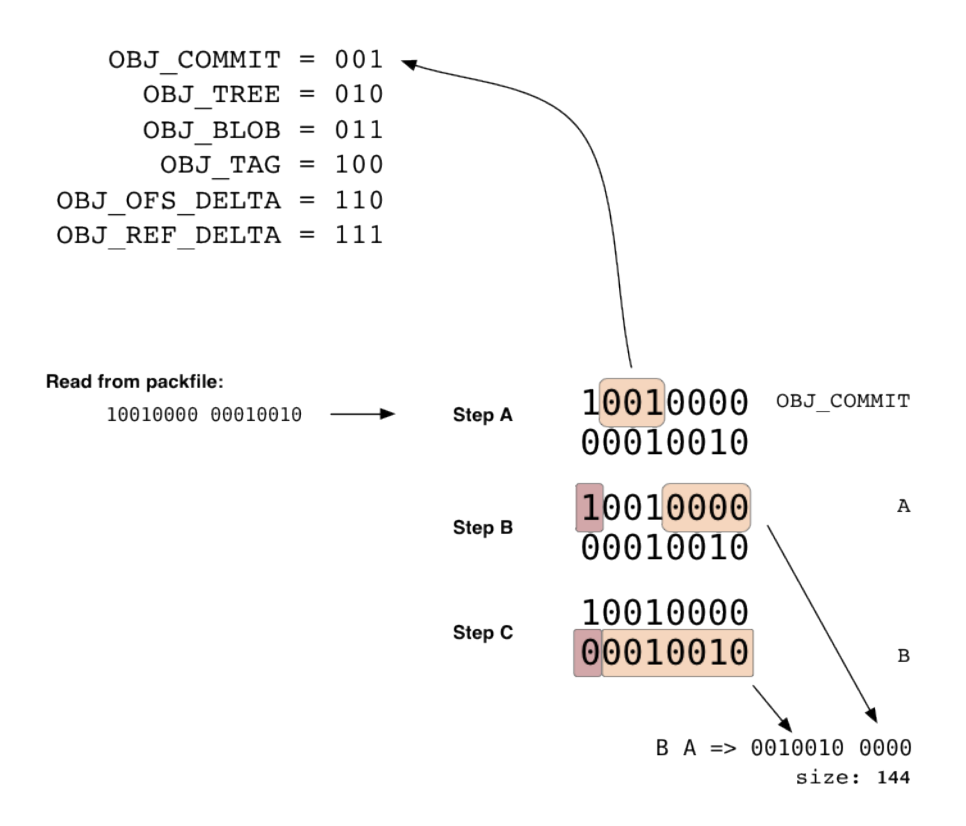
Après ça, vous avec une suite d'objets packagés, ordonnés par leur SHA, chacun représenté par l'entête de l'objet et son contenu. Une somme SHA de 20 bits de tous les SHAs du packfile se trouve dans les 20 bits à la fin du fichier.


objects/pack/pack-4eb8b...c5.pack
"PACK"
4b
version
4b
packfile version number
entries
4b
entries in the packfile
Offsets from pack idx
offset
8 bits
1
type(3)
size(4) A
1b
1
size(7) B
1b
0
size(7) C
1b
data
(C << 11) & (B << 4) & A bytes when expanded
offset
1
type(3)
size(4) A
1b
1
size(7) B
1b
1
size(7) C
1b
0
size(7) D
1b
data
(D << 18) & (C << 11) & (B << 4) & A bytes when expanded
...
...
checksum SHA1
20b

L'entête de l'objet est une série de un ou plusieurs morceaux d'octets (8 bits) qui définissent le type d'objet représenté par les données qui suivent, et la taille de l'objet quand il est décompressé. Chaque octet ne contient réellement que 7 bits de données, le premier bit ne disant que si cet octet est le dernier avant que ne commence les données. Si le premier bit est 1, vous pourrez continuer à lire le prochain octet. sinon les données commencent au prochain octet. Les 3 premiers bits du premier octet spécifient le type de données, en accord avec la table ci-dessous.

(Actuellement, sur les 8 valeurs qui peuvent être exprimées avec 3 bits (0-7), 0 (000) est 'undefined' et 5 (101) n'est pas encore utilisée.)

Nous pouvons voir ici un exemple d'entête de 2 octets, où le premier spécifie que les données suivantes sont un commit, et le le reste des bits du premier octet accompagnés de ceux du second octet spécifient que les données feront 144 octets une fois décomprimés.



Il est important de savoir que la taille spécifiée dans les données de l'entête n'est pas la taille des données qui suivent, mais la taille des données *une fois décomprimés*. C'est pourquoi les offset de l'index du packfile sont si utiles, sinon vous devriez décompresser chaque objet juste pour savoir quand la prochaine entête commence.

La partie donnée n'est qu'un stream zlib pour des types d'objet non-delta; pour la représentation de 2 objets delta, la portion de donnée contient quelque chose qui identifie de quel objet de base dépend cet objet delta, et le delta à appliquer sur l'objet de base pour retrouver cet objet. `ref-delta` utilise un hash de 20 octets pour l'objet de base au début des données, alors que `ofs-delta` stocke un offset à l'intérieur du même packfile pour identifier l'objet de base. Dans chacun des cas, vous devez suivre 2 contraintes importantes si vous voulez ré-implémenter ce fonctionnement:

- la représentation du delta doit être basée sur quelqu'autre objet à l'intérieur du même packfile;
- l'objet de base doit être de même type sous-jacent (blob, tree, commit ou tag);

RAW GIT

Here we will take a look at how to manipulate git at a more raw level, in case you would like to write a tool that generates new blobs, trees or commits in a more artificial way. If you want to write a script that uses more low-level git plumbing to do something new, here are some of the tools you'll need.

Creating Blobs

Creating a blob in your Git repository and getting a SHA back is pretty easy. The git hash-object command is all you'll need. To create a blob object from an existing file, just run it with the '-w' option (which tells it to write the blob, not just compute the SHA).

```
$ git hash-object -w myfile.txt
6ff87c4664981e4397625791c8ea3bbb5f2279a3

$ git hash-object -w myfile2.txt
3bb0e8592a41ae3185ee32266c860714980dbed7
```

The STDOUT output of the command will the the SHA of the blob that was created.

Creating Trees

Now lets say you want to create a tree from your new objects. The git mktree command makes it pretty simple to generate new tree objects from git ls-tree formatted output. For example, if you write the following to a file named '/tmp/tree.txt' :

```
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    file1
100644 blob 3bb0e8592a41ae3185ee32266c860714980dbed7    file2
```

and then piped that through the git mktree command, Git will write a new tree to the object database and give you back the new sha of that tree.

```
$ cat /tmp/tree.txt | git mk-tree
f66a66ab6a7bfe86d52a66516ace212efa00fe1f
```

Then, we can take that and make it a subdirectory of yet another tree, and so on. If we wanted to create a new tree with that one as a subtree, we just create a new file (/tmp/newtree.txt) with our new SHA as a tree in it:

```
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    file1-copy
040000 tree f66a66ab6a7bfe86d52a66516ace212efa00fe1f    our_files
```

and then use git mk-tree again:

```
$ cat /tmp/newtree.txt | git mk-tree
5bac6559179bd543a024d6d187692343e2d8ae83
```

And we now have an artificial directory structure in Git that looks like this:

```
.
|-- file1-copy
`-- our_files
    |-- file1
    `-- file2

1 directory, 3 files
```

without that structure ever having actually existed on disk. Plus, we have a SHA (5bac6559) that points to it.

Rearranging Trees

We can also do tree manipulation by combining trees into new structures using the index file. As a simple example, let's take the tree we just created and make a new tree that has two copies of our 5bac6559 tree in it using a temporary index file. (You can do this by resetting the GIT_INDEX_FILE environment variable or on the command line)

First, we read the tree into our index file under a new prefix using the git read-tree command, and then write the index contents as a tree using the git write-tree command:

```
$ export GIT_INDEX_FILE=/tmp/index
$ git read-tree --prefix=copy1/  5bac6559
$ git read-tree --prefix=copy2/  5bac6559
$ git write-tree
bb2fa6de7625322322382215d9ea78cfe76508c1

$>git ls-tree bb2fa
040000 tree 5bac6559179bd543a024d6d187692343e2d8ae83    copy1
040000 tree 5bac6559179bd543a024d6d187692343e2d8ae83    copy2
```

So now we can see that we've created a new tree just from index manipulation. You can also do interesting merge operations and such in a temporary index this way - see the git read-tree docs for more information.

Creating Commits

Now that we have a tree SHA, we can create a commit object that points to it. We can do this using the git commit-tree command. Most of the data that goes into the commit has to be set as environment variables, so you'll want to set the following:

```
GIT_AUTHOR_NAME
GIT_AUTHOR_EMAIL
GIT_AUTHOR_DATE
GIT_COMMITTER_NAME
GIT_COMMITTER_EMAIL
GIT_COMMITTER_DATE
```

Then you will need to write your commit message to a file or somehow pipe it into the command through STDIN. Then, you can create your commit object based on the tree sha we have.

```
$ git commit-tree bb2fa < /tmp/message
a5f85ba5875917319471dfd98dfc636c1dc65650
```

If you want to specify one or more parent commits, simply add the shas on the command line with a '-p' option before each. The SHA of the new commit object will be returned via STDOUT.

Updating a Branch Ref

Now that we have a new commit object SHA, we can update a branch to point to it if we want to. Lets say we want to update our 'master' branch to point to the new commit we just created - we would use the git update-ref command:

```
$ git update-ref refs/heads/master a5f85ba5875917319471dfd98dfc636c1dc65650
```

LES PROTOCOLES DE TRANSFERT

Nous allons voir ici les communications qui ont lieu entre les client et les serveur pour permettre à Git de transférer des données.

Récupération des Données avec HTTP

Git utilise un protocole un peu plus fragile quand il récupère des données sur une adresse http/https. Dans ce cas, toute la logique se déroule du côté client. Le serveur ne requiert pas de configuration spéciale - n'importe quel serveur web statique fonctionnera si le dossier git que vous récupérez est dans un chemin accessible depuis le serveur web.

Afin de faire fonctionner tout ça, vous devez lancer une simple commande sur le dépôt du serveur chaque fois que quelque chose est mis à jour. git update-server-info met à jour les objects/info/packs et les fichiers d'info/refs pour afficher quelles références et packfiles sont disponibles, puisque vous ne pouvez pas faire une liste avec http. Quand cette commande est lancée, le fichier objects/info/packs ressemblera à quelque chose comme ça:

```
P pack-ce2bd34abc3d8ebc5922dc81b2e1f30bf17c10cc.pack
P pack-7ad5f5d05f5e20025898c95296fe4b9c861246d8.pack
```

Donc si la récupération (fetch) ne peut pas trouver un fichier seul, il peut essayer de le chercher dans ces packfiles. Le fichier info/refs ressemblera à quelque chose comme ça:

```
184063c9b594f8968d61a686b2f6052779551613    refs/heads/development
32aae7aef7a412d62192f710f2130302997ec883    refs/heads/master
```

Quand vous faite une récupération (fetch) sur ce dépôt, elle commencera par ces références et parcourera les objets commits jusqu'à que le client ait tous les objets qu'il a besoin.

Par exemple, si vous demandez de récupérer la branche "master", il verra que "master" pointe sur `32aae7ae` et que votre master pointe sur `ab04d88`, donc vous aurez besoin de `32aae7ae`. Vous récupérez cet objet:

```
CONNECT http://myserver.com
GET /git/myproject.git/objects/32/aae7aef7a412d62192f710f2130302997ec883 - 200
```

et il ressemble à ça:

```
tree aa176fb83a47d00386be237b450fb9dfb5be251a
parent bd71cad2d597d0f1827d4a3f67bb96a646f02889
author Scott Chacon <schacon@gmail.com> 1220463037 -0700
committer Scott Chacon <schacon@gmail.com> 1220463037 -0700

added chapters on private repo setup, scm migration, raw git
```

donc maintenant il récupère le "tree" `aa176fb8`:

```
GET /git/myproject.git/objects/aa/176fb83a47d00386be237b450fb9dfb5be251a - 200
```

qui ressemble à ça:

```
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING
100644 blob 97b51a6d3685b093cfb345c9e79516e5099a13fb    README
100644 blob 9d1b23b8660817e4a74006f15fae86e2a508c573    Rakefile
```

donc il récupère ces objets:

```
GET /git/myproject.git/objects/6f/f87c4664981e4397625791c8ea3bbb5f2279a3 - 200
GET /git/myproject.git/objects/97/b51a6d3685b093cfb345c9e79516e5099a13fb - 200
GET /git/myproject.git/objects/9d/1b23b8660817e4a74006f15fae86e2a508c573 - 200
```

Curl est utilisé pour cette opération, et plusieurs thread parallèles sont ouverts pour accélerer ce processus. Quand il a terminé d'examiner le tree pointé par le commit, il récupère le prochain parent:

```
GET /git/myproject.git/objects/bd/71cad2d597d0f1827d4a3f67bb96a646f02889 - 200
```

Maintenant dans ce cas, le commit qui revient ressemble à ça:

```
tree b4cc00cf8546edd4fcf29defc3aec14de53e6cf8
parent ab04d884140f7b0cf8bbf86d6883869f16a46f65
author Scott Chacon <schacon@gmail.com> 1220421161 -0700
committer Scott Chacon <schacon@gmail.com> 1220421161 -0700

added chapters on the packfile and how git stores objects
```

et vous pouvez voir que le parent, `ab04d88`, est là où pointe la branche courante. Donc, nous avec récupérer récursivement ce tree puis arrêter puis que nous avons récupérer tout ce qui nous manquait jusqu'à ce point. Vous pouvez forcer Git à verifier par deux fois que nous avons tout ce qu'il nous avec l'option '--recover'. Voir git http-fetch pour plus d'informations.

Si la récupération de l'un des objet seul échoue, Git téléchargement automatiquement les index des packfiles pour retrouver le sha nécessaire, puis téléchargera ce packfile.

Si vous avec un serveur git qui fournit des dépôt de cette façon, il est important d'implémenter un hook de post-receive qui exécute la commande 'git update-server-info' à chaque mise à jour pour éviter les confusions.

Récupération des Données avec Upload Pack

Pour les protocoles plus intelligents, la récupération d'objets est bien plus efficace. Un socket est ouvert, soit par ssh ou sur le port 9418 (dans ce cas avec le protocole git://), et la commande git fetch-pack sur le client commence à communiquer avec un processus forké de la commande git upload-pack sur le serveur.

Puis le serveur va dire au client quels SHAs il a pour chaque référence et le client devinera ce qu'il à besoin et lui répondra avec la liste des SHAs qu'il veut et de ceux qu'il a déjà.

A ce moment, le serveur va générer un packfile avec tous les objets que le clients a besoin et va commencer à l'envoyer en stream au client.

Regardons un exemple.

Le client se connecte et envoie l'entête de requête. La commande de clonage

```
$ git clone git://myserver.com/project.git
```

produit la requête suivante:

```
0032git-upload-pack /project.git\000host=myserver.com\000
```

Les 4 premiers bits contiennent la longueur hexadécimale de la ligne (en incluant ces 4 premiers bits et le retour-chariot s'il est présent). Ensuite viennent la commande et les arguments. Suivi par un bit nul puis les informations du host. La requête se termine par un bit nul.

Sur le serveur, la requête est analysées et transformée en un appel à git-upload-pack:

```
$ git-upload-pack /path/to/repos/project.git
```

Ceci retourne immédiatement les informations du dépôt:

```
007c74730d410fcb6603ace96f1dc55ea6196122532d HEAD\000multi_ack thin-pack side-band side-band-64k ofs-delta shall
003e7d1665144a3a975c05f1f43902ddaf084e784dbe refs/heads/debug
003d5a3f6be755bbb7deae50065988cbfa1ffa9ab68a refs/heads/dist
003e7e47fe2bd8d01d481f44d7af0531bd93d3b21c01 refs/heads/local
003f74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/master
0000
```

Chaque ligne commence par 4 bits représentant la longueur de la déclaration en hexadécimal. La section se termine par un déclaration de longueur de ligne de 0000.

Ceci est renvoyé comme-tel au client. Le client répond avec une autre requête:

```
0054want 74730d410fcb6603ace96f1dc55ea6196122532d multi_ack side-band-64k ofs-delta
0032want 7d1665144a3a975c05f1f43902ddaf084e784dbe
0032want 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a
0032want 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01
0032want 74730d410fcb6603ace96f1dc55ea6196122532d
00000009done
```

C'est envoyé au processus git-upload-pack encore ouvert qui envoie ensuite en stream la réponse finale:

```
"0008NAK\n"
"0023\002Counting objects: 2797, done.\n"
"002b\002Compressing objects:   0% (1/1177)   \r"
"002c\002Compressing objects:   1% (12/1177)   \r"
"002c\002Compressing objects:   2% (24/1177)   \r"
"002c\002Compressing objects:   3% (36/1177)   \r"
"002c\002Compressing objects:   4% (48/1177)   \r"
"002c\002Compressing objects:   5% (59/1177)   \r"
"002c\002Compressing objects:   6% (71/1177)   \r"
```

```
"0053\002Compressing objects:   7% (83/1177)   \rCompressing objects:   8% (95/1177)   \r"
...
"005b\002Compressing objects: 100% (1177/1177)   \rCompressing objects: 100% (1177/1177), done.\n"
"2004\001PACK\000\000\000\002\000\000\n\355\225\017x\234\235\216K\n\302"...
"2005\001\360\204{\225\376\330\345]z2673"...
...
"0037\002Total 2797 (delta 1799), reused 2360 (delta 1529)\n"
...
"<\276\255L\273s\005\001w0006\001[0000"
```

Voir le chapitre précédent sur le Packfile pour plus d'information sur le format de donnée du packfile présent dans la réponse.

Publier des Données

Publier des données sur les protocoles git et ssh est similaire mais plus simple. En gros, le client demande une instance de receive-pack, qui est lancé si le client y a accès, puis le serveur renvoie une nouvelle fois tous les SHAs de référence et le client génère un packfile pour tout ce dont le serveur a besoin (généralement seulement si ce qui se trouve sur le serveur est un ancêtre direct de ce qui doit être publié) et envoie ce packfile en stream vers le serveur, où le serveur peut le stocker sur le disque et construire son index, ou le déballer (unpack) s'il contient beaucoup d'objets.

L'intégralité de ce processus est réalisé par la commande git send-pack sur le client, qui est appelée par git push et la commande git receive-pack du côté du serveur, qui est appelée par le processus de connexion ssh ou le daemon git (dans le cas d'un serveur ouvert à la publication).